UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

2021

Trabajo fin de grado

Aprendizaje a Gran Escala Mediante Procesos Gaussianos Usando Pytorch y GPUs

David García Fernández

Campus Internacional excelencia UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aprendizaje a Gran Escala Mediante Procesos Gaussianos Usando Pytorch y GPUs

Autor: David García Fernández
Tutor: Daniel Hernández Lobato

mayo 2021

David García Fernández
Aprendizaje a Gran Escala Mediante Procesos Gaussianos Usando Pytorch y GPUs

David García Fernández

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Agradecimientos

Mucho ha llovido desde que terminé la última asignatura de la carrera y desde que le eché el ojo a este trabajo. Para completarlo, el camino no ha sido siempre fácil, pero si gratificante, por todo lo que ha aportado a mi crecimiento tanto como persona, como estudiante. Por supuesto, para que esto haya sido posible, el apoyo que he recibido de tantas personas ha sido esencial.

Por su contribución y ayuda, por estar siempre disponible para mi, quería mostrar mi agradecimiento a mi tutor, Daniel Hernández Lobato.

A mi novia, Laura, por estar a mi lado desde antes del principio, y por esperar con más ilusión que yo, este final. A mi familia, que algo habrá hecho, digo yo, para que hoy pueda estar aquí.

No podía olvidarme del grupo de los *joumis*. A Ángel, por nuestras conversaciones sobre matemáticas, IA y Linux. Ahora que he terminado te debo un Growler, amigo. A Galán, por esas noches locas. A Sergio que se fue de este país y me juro que no volvería hasta que terminase este trabajo. A John, con quien tanto tiempo pasé en la biblioteca de la EPS. A Gus, por las conversaciones sobre hardware. A Aemuv, que le perdí la pista cuando se fue a comprar tabaco y a Charlie, que le siguió. A todos, gracias.

Finalmente, a todos mis profesores, que desde mi más tierna infancia, han hecho posible que hoy pueda pensar, cuestionar y aprender. Nunca olvidare esta deuda, y espero poder pagársela a las siguientes generaciones.

Resumen

A lo largo de este trabajo, hablaremos del aprendizaje supervisado mediante regresión para realizar inferencia. En concreto, utilizando procesos gaussianos. El problema, es que este tipo de modelo escala muy mal a medida que aumenta el número de datos sobre el que trabajamos; en particular, y con N el número de datos de entrenamiento, tiene un coste computacional de $O(N^3)$ y $O(N^2)$ de consumo de memoria. Por esta razón introduciremos un marco que nos permita aproximar nuestro modelo original y reducir el coste a la vez que mantenemos un buen ajuste.

Las técnicas que vamos a utilizar, basadas en los trabajos de Snelson y Ghahramani [1] y Michelis k. Titsias [2], enfocan el problema mediante la inferencia variacional. Es decir, trabajaremos con la distribución de probabilidad de un proceso gaussiano ampliada con un conjunto de variables latentes [1]. De esta distribución derivaremos una cota inferior del logaritmo de la verosimilitud marginal, la cual, nos servirá para aproximar nuestras variables latentes a la distribución original a la vez que optimizamos el resto de los hiperparámetros [2]. Al final obtendremos un coste temporal de $O(NM^2)$ y $O(NM)$ de memoria [2], donde M se corresponde con el número de puntos inductores y, en general, es mucho menor que N. No obstante, estos costes -especialmente el de la memoria-, siguen estando por encima de lo deseable para entornos con grandes cantidades de datos. Afortunadamente, Hensman et al. [3], nos ofrecen como solución la optimización estocástica. De esta forma podremos segmentar el conjunto total de datos en mini-batches, cuyo procesamiento nos permitirá acercarnos, paso a paso, a la función objetivo y obtener buenos resultados antes incluso de procesar las N instancias de datos.

Toda esta algoritmia la implementaremos utilizando la biblioteca Pytorch. Esta nos facilitará una interfaz para: trabajar con tensores, abstraer el cálculo de los gradientes (optimización) y el computo de nuestros algoritmos en GPU. Finalmente, veremos de qué forma afectan los diferentes acercamientos a los problemas planteados. En concreto, como se ajustan nuestros diferentes modelos a los datos de acuerdo a métricas como el error cuadrático medio o el logaritmo de la verosimilitud. Además, también comprobaremos como va mejorando el rendimiento a medida que implementamos los nuevos métodos. Esto lo haremos utilizando conjuntos de datos procedentes de repositorio de datos UCI [4].

Palabras clave

Proceso Gaussiano, variables latentes, Maximización de la Esperanza, Inferencia variacional, Divergencia de Kullback-Leibler, inferencia bayesiana, verosimilitud, Pytorch, GPU, rendimiento, regresión, Kernel, RBF, ajuste, cota mínima, entropía relativa, Python, Cuda, optimización.

Abstract

Along this work, we will be talking about inference by using supervised learning regression, and, as the title implies, we will be having a Gaussian Processes as our model. Nonetheless, this model is analytically intractable for large datasets. The reason behind this is that, given an input dataset of size N, time complexity scales as $O(N^3)$, and storage as $O(N^2)$ [2]. In order to overcome this issue, we introduce a framework that lets us approximate our original model and reduce our cost while getting a good fit for our data.

The techniques that we are going to use are based on the papers of Snelson & Ghahramani [1] and Titsias [2]. To sum them up, we will augment the Gaussian Process probability distribution with a set of latent variables. As a consequence, we can get a lower bound of the original log marginal likelihood that we can use to get the optimized hyperparameters along with the set of latent variables which approximates better to the original model. In the end, we manage to reduce the cost by reducing the use of the complete dataset. Specifically, we get a computational complexity of $O(MN^2)$, and $O(MN)$ for the storage [2]; that is, given N as the size of the dataset and M the size of the inducing point's set. However, despite all of that, the model continues to be intractable for larger datasets, specifically in memory usage [3]. Affortunately, Hensman, Fusi & Lawrence [3], propose a solution in which we can split the whole dataset in mini-batches that we can process, step by step, in order to reach the solution. Additionally, this method let us get good results even before processing the N instances of data.

To implement all these algorithms we will use Python's library Pytorch, which will provide us an interface to operate with the GPU, tensors and abstract gradient's calculus when using stochastic optimization. After the implementation, we will be able to test how different approaches (techniques, models and hardware resources) affect our results. In particular, we will see how our different models fit the data according to metrics such as Mean Squared Error and Log Likelihood. Furthermore, we will check how performance improves when using the aforementioned methods in an environment established by some real datasets (YearPredictionMSD, Physicochemical...) provided by the UCI Repository [4].

Keywords

Gaussian Process, Sparse Gaussian Process, latent variables, Expectation Maximization, EM, Variational Inference, KL, Kullback-Leibler Divergence, Bayesian Inference, Likelihood, Pytorch, GPU, performace, regression, Kernel, RBF, fit, elbo, evidence lower bound, pseudo-inputs, inducing set, relative entropy, Python, Cuda, Adam, optimization.

ÍNDICE

Listas

Lista de ecuaciones

Lista de figuras

Lista de tablas

.

1

INTRODUCCIÓN

1.1. Motivación.

En la actualidad estamos viendo como las tecnologías relacionadas con el aprendizaje automático están viviendo un auténtico auge. En esto, por supuesto, influye el gran abanico de posibilidades y facilidades que nos proporcionan, ya sea para: la realización de estudios genéticos, inversiones en los mercados financieros, o simplemente un sistema de recomendación de películas. Todas estas posibilidades, y muchas más, tienen como denominador común el descubrimiento y explotación de las relaciones que existen entre los datos.

En general, la literatura que se encarga de abordar estos temas, suele distinguir entre tres corrientes para afrontar los diferentes retos [6]. De esta forma, encontramos el aprendizaje no supervisado, utilizado para encontrar relaciones y patrones entre los datos y agruparlos según sus características [6]. Por otro lado, tenemos el aprendizaje por refuerzo, que consiste en encontrar el suceso o serie, que permita maximizar la probabilidad de que ocurra el objetivo definido por el sistema [1]. Y, finalmente, el aprendizaje supervisado, caracterizado por mapear un conjunto de datos de entrada, a un conjunto de datos de salida. Cuando el dominio de las salidas es un conjunto finito de valores nominales, se dice que es un problema de *clasificación*. Si, por el contrario, el dominio de las salida son los números reales, se dice que es un problema de *regresión* [6], y será al que nos enfrentemos a lo largo de este trabajo.

En este momento entran en escena los procesos gaussianos (distribución conjunta gaussiana), los cuales, nos permiten modelar de forma flexible, entornos de alta dimensionalidad compuestos por variables aleatorias que se distribuyen de forma gaussiana. La fórmula elegida para realizar esta tarea es la regresión lineal bayesiana. Es decir, trataremos de encontrar una función f, que nos permita relacionar el conjunto de entrada ($\mathbf{x}$), con el de salida ($\mathbf{y}$) mediante $y = f(x) + \epsilon$, donde ϵ es el ruido gaussiano asociado [7]. Esto nos permitirá predecir, cuando tengamos datos de entrada conocidos, las salidas desconocidas asociadas a dicha entrada con un determinado grado de confianza. Entrando algo más en detalle, el mecanismo sobre el que vamos a asentar la búsqueda de nuestra f, va a ser

[1] A este *modus operandi* también se le conoce como maximizar la señal de recompensa [10].

el teorema de bayes, en el que pondremos un proceso gaussiano como prior sobre funciones, para conseguir el posterior sobre las mismas [6]. Esto dará como resultado el modelo flexible para entornos de alta dimensionalidad que comentábamos antes.

Sin embargo, no todo son virtudes, y algo tan esencial como disponer de datos para nuestro problema, puede convertirse en nuestra espada de Damocles. La razón es que los datos son un recurso del que cada vez tenemos más en cantidad y variedad, nótese que solo en este momento se están generando casi 10000 tweets por segundo [11]; cada uno con su autor, menciones y etiquetas; cada uno con su kit completo de metadatos.

El problema radica en cómo afecta esto al rendimiento de nuestro modelo. En particular, para entradas de tamaño N encontramos costes computacionales de $O(N^3)$ y costes de almacenamiento de $O(N^2)$. No obstante, a lo largo de los últimos años, algunos autores [1] [2] [3], han agrupado y desarrollado un conjunto de técnicas que nos permiten utilizar un subconjunto reducido de datos para aproximar la distribución posterior obtenida con la regla de bayes. Esto nos permitirá reducir el coste a la vez que mantenemos un buen ajuste del modelo a los datos. En concreto, la solución propuesta nos permitirá obtener costes computacionales por debajo de $O(MN^2)$, y por debajo de $O(N^2)$ para el almacenamiento en memoria. Al mismo tiempo, el desarrollo de sistemas software y hardware capaces de paralelizar y trabajar con grandes conjuntos de datos, ha hecho posible reducir incluso más nuestro coste temporal.

Nuestra motivación parte, precisamente, de estas ideas. Queremos desarrollar las diferentes técnicas, probarlas en sistemas con distintas prestaciones, y ver el impacto que tienen en algunos de los escenarios reales recogidos en el repositorio UCI [4].

1.2. Objetivos.

El propósito de este trabajo consiste en observar cómo escala el coste temporal de un proceso gaussiano y su distribución posterior aproximada cuando realizamos inferencia. Dado N el número de instancias de la entrada y M el de puntos inductores; las soluciones teóricas nos hablan de costes computacionales de $O(N^3)$ y de almacenamiento de $O(N^2)$) para el proceso gaussiano, mientras que su versión aproximada mediante optimización estocástica nos permitirá obtener buenos resultados antes de $O(MN^2)$ y $O(MN)$ respectivamente. En general, buscaremos $M << N$, donde M será un parámetro libre que nos permita ajustar el coste a la capacidad computacional del sistema que se vaya a utilizar.

Por otro lado, dado que también nos interesa tener un buen sistema de predicciones, gran parte del trabajo orbitará en torno a la selección de hiperparámetros para que nuestro modelo se ajuste a los datos. Por esta razón será importante desarrollar un sistema de métricas que nos permita juzgar la calidad del ajuste. Finalmente, también nos interesará conocer el impacto de diferentes sistemas

hardware (CPU/GPU) en el rendimiento de las soluciones propuestas. Con esta finalidad, incluiremos como objetivo, el aprendizaje y utilización de tecnologías que nos ofrecen este tipo de capacidades, en este caso la librería Pytorch.

1.3. Organización de la memoria.

La memoria de este trabajo está estructurada de la siguiente forma:

1.– Capítulo 1: Introducción. En este capítulo expondremos los conceptos iniciales, la motivación y los objetivos del trabajo.

2.– Capítulo 2: Estado del arte. En este capítulo se establecen los fundamentos teóricos para realizar inferencia utilizando procesos gaussianos. Además, veremos que técnicas podemos usar para reducir el coste temporal y de almacenamiento. EL cierre del capítulo se consumará con la introducción de las tecnologías que vamos a utilizar para la implementación.

3.– Capítulo 3: Diseño. El objetivo del capítulo 3 es definir y modelar los detalles de nuestra implementación. El enfoque que vamos a adoptar, y del que nace nuestro diseño, consiste en la creación de un conjunto de tests para probar y medir los costes de nuestros algoritmos. Dado que nuestra extensión es limitada y la carga de trabajo se centra en otras áreas no abordaremos esta sección como un estricto proyecto de software, sino como una forma flexible para guiarnos durante el desarrollo.

4.– Capítulo 4: Desarrollo. El fin de este capítulo es concretar detalles de la implementación que se escapan del ámbito del diseño. Es decir, pasamos de describir qué hacer a cómo hacerlo.

5.– Capítulo 5: Integración, pruebas y resultados. Este capítulo comienza detallando los requisitos del sistema que son necesarios para replicar las pruebas que hemos hecho, posteriormente pasaremos a exponer los tests que hemos diseñado y los resultados que generan.

6.– Capítulo 6: Conclusiones y trabajo futuro. Finalmente, en este capítulo analizaremos los resultados para determinar cuánto se ajustan a los planteamientos teóricos y de donde surgen las discrepancias con los mismos. Además, indicaremos cuales son algunas de las áreas que merecen hacerse notar para una posible extensión del trabajo.

2

Estado del Arte

2.1. Procesos Gaussianos.

Un proceso gaussiano o GP, como veremos en detalle más adelante, es una generalización de la distribución gaussiana para un número finito de variables aleatorias [12]. Por esta razón, resulta apropiado empezar a desgranar este tema desde esta distribución.

La distribución gaussiana es un modelo matemático ampliamente utilizado a la hora de modelar la distribución de variables aleatorias continuas [13] [6] [5]. Esto lo hace mediante la función de densidad dada por la ecuación 2.1 en el caso univariante -con μ y σ media y varianza respectivamente-, y la ecuación 2.2 en el caso multivariante -en este caso con μ y Σ vector de medias y matriz de covarianzas de tamaño D y $D \times D$ respectivamente-.

$$\mathcal{N}(\mathbf{x}|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \tag{2.1}$$

Una de las principales razones que justifican su uso, es por su capacidad para modelar una gran variedad de escenarios diferentes. Por ejemplo, atendiendo al teorema central del límite [6], podemos ver como distribuciones como la binomial convergen, a medida que aumentan los sucesos observados, en la distribución gaussiana [14]. Además, no es raro encontrarse esta circunstancia en la realidad, y de hecho, esta distribución es especialmente útil a la hora de modelar el error existente en los datos [14].

$$\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{D/2}\,|\Sigma|^{1/2}}} exp\left\{-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right\} \tag{2.2}$$

Otra de las razones por la que se usa tanto es por la carga de información que contiene esta distribución, y es que, dadas una media y una varianza, maximiza la entropía [5] [14]. Además, solo con estos dos parámetros podemos capturar muy bien las propiedades de la distribución a la vez que sigue siendo muy manejable.

Ahora que hemos introducido nuestro concepto base, ha llegado la hora de introducir los procesos gaussianos. En concreto, y más formalmente: un Proceso Gaussiano es una colección de variables

aleatorias, de las cuales, cualquier conjunto finito elegido tiene una distribución gaussiana conjunta y consistente [12] [7]. Esto supone una distribución de probabilidad sobre funciones [6]. La potencia de este concepto radica en el conjunto de reglas y propiedades de la distribución gaussiana, por ejemplo la de la marginalización es la que nos permite estudiar un subconjunto finito de variables sin necesidad de tener que analizar el conjunto infinito al que pertenecen [12].

En definitiva, un proceso gaussiano no se encarga de describir el comportamiento de una variable aleatoria, si no de gobernar las propiedades de un conjunto de ellas [1] [12]. Estas propiedades [15] nos van a ayudar a efectuar operaciones en forma cerrada con el GP. Este es uno de los atractivos de la distribución gaussiana, ya que, dentro del marco de la regresión lineal bayesiana, será suficiente para poder obtener y evaluar nuestros objetivos de forma analítica y que la solución conserve las mismas propiedades [2] [15] [5]. Como consecuencia, el trabajo se simplifica con respecto a otro tipo de distribuciones que incluso pueden dar lugar a expresiones no computables.

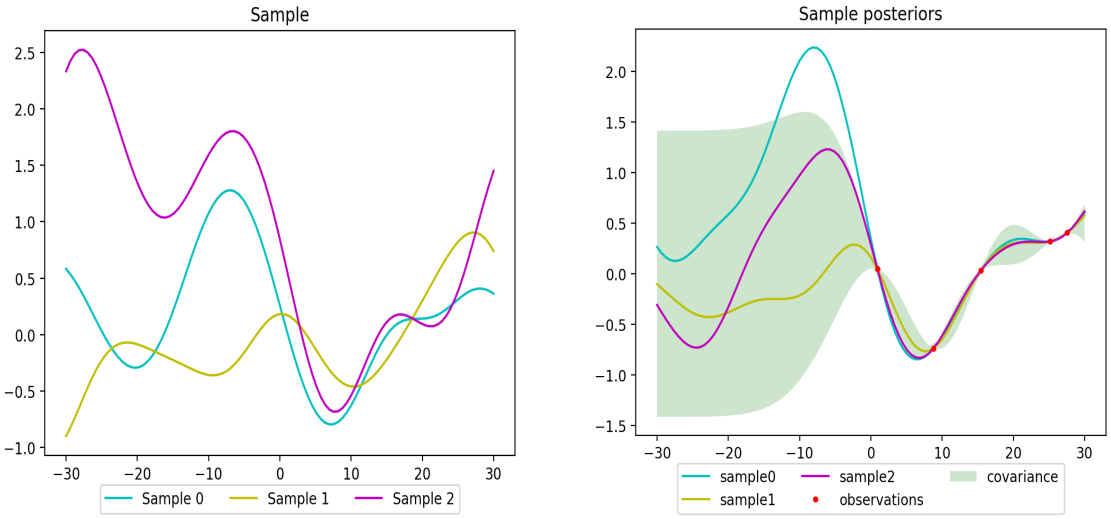
Hace apenas un momento, introducíamos la regresión lineal bayesiana. Esta es una técnica muy extendida basada en la regla de Bayes (apéndice A.1, con la que, dadas unas observaciones sobre nuestro problema, podemos afinar más el modelo probabilístico. La única pega que, a priori, podemos verle, es la falta de flexibilidad que nos ofrece una función lineal en entornos multi-dimensionales. No obstante, como exponen Rassmusen & Williams en su trabajo [12], podemos sortear este problema tomando una función ϕ que abstraiga nuestra entrada de un espacio de características de alta dimensionalidad. El resultado es que conseguimos trabajar en un espacio proyectado, manteniendo la linealidad (con respecto a la entrada), y el espacio de trabajo con expresiones analíticamente computables.

En regresión nuestro objetivo va a ser encontrar una función o variable latente f que para una entrada -o vector de características- x nos devuelva su observación y asociada, o en otras palabras, que cumpla $y = f(x) + \epsilon$, donde ϵ es el ruido que, asumimos, tiene una distribución gaussiana $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$ [8]. Por lo general, trabajaremos con conjuntos de estos datos usando la notación $\mathbf{y} = (y_1, ..., y_N)^T$, $\mathbf{f} = f(f_1(x_1), ..., f_N(x_N)))$ y $\mathbf{X} = (x_1, ...x_N)^T$, respectivamente. Por otro lado, cuando queramos indicar que estamos trabajando con funciones y observaciones desconocidas dada una entrada, (caso predictivo), utilizaremos * como subíndice; y para el conjunto de los hiperparámetros θ. Finalmente, solo falta añadir que cuando trabajemos en entornos de alta dimensionalidad estaremos haciendo que $f(x) = \phi(x)^T\mathbf{w}$, donde $\mathbf{w}$ es un hiperplano [8], aunque por claridad, omitiremos esto de nuestra notación. En la figura 2.1, podemos ver como diferentes muestras dibujadas a partir del prior se ajustan a los datos observados.

Si queremos aplicar esta regla, lo primero que tenemos que hacer es establecer nuestro prior sobre las variables latentes $(\mathbf{f}, \mathbf{f}_*)$, para ello vamos a utilizar un proceso gaussiano como el de la ecuación

[1] Esto, como se ilustra en la figura 2.1, se hace en forma de proceso estocástico.

[2] Atendiendo a dichas propiedades, un proceso gaussiano no deja de ser una distribución gaussiana.



(a) Muestras del prior del GP.

(b) Muestras del posterior del GP.

Figura 2.1: En las subfiguras 2.1(a) y 2.1(b) podemos ver las muestras de funciones dibujadas a partir del prior y el posterior de un proceso gaussiano. Estas figuras han sido generadas mediante el código presente en el apéndice B.5.

2.3 [6] [7]. Las propiedades que gobiernan nuestro proceso son la media [3] $m(x) = \mathrm{E}[f(x)]$, y la función de covarianzas o *kernel* $k(x, x')$ calculada sobre los puntos de entrada de forma que podamos configurar una matriz $\mathbf{K}$, de tamaño NxN, y de forma que $\mathbf{K_{i,j}} = k(i, j)$ [6].

$$\begin{aligned} f(x) &\sim \mathcal{GP}(m(x), k(x, x')) \\ p(\mathbf{f}) &= \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K}) \end{aligned} \tag{2.3}$$

Para la función de *verosimilitud* o *likelihood* (parte izquierda del numerador de la regla de Bayes A.1), debemos tener en cuenta que un proceso gaussiano no deja de ser una distribución normal conjunta. La expresión se puede ver en la ecuación 2.4, donde calcularemos la probabilidad conjunta de todos los puntos observados.

$$p(\mathbf{y}|\mathbf{f}) = \prod_{i=1}^{N} p(y_i|f_i) = \mathcal{N}(\mathbf{y}|\mathbf{f}, \sigma_n^2 I) \tag{2.4}$$

El último factor que necesitamos para poder obtener nuestro posterior, es la *verosimilitud marginal*, que podemos conseguir integrando (marginalizando) las variables latentes sobre nuestra función 2.5 de likelihood.

$$p(\mathbf{y}) = \int p(\mathbf{y}|\mathbf{f})p(\mathbf{f})d\mathbf{f} = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}) + \sigma_n^2 I) \tag{2.5}$$

[3] Dada la flexibilidad del proceso gaussiano para modelar la media, se suele tomar $m(X) = 0$ para el prior [6].

Finalmente, empleando la regla de Bayes, obtenemos el posterior $p(\mathbf{f}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{f})p(\mathbf{f})}{p(\mathbf{y})}$. Si usamos la entrada $\mathbf{X}_*$ asociada a las $\mathbf{f}_*$ que queremos inferir, estaremos hablando de la distribución posterior predictiva $\mathbf{f}_* \sim \mathcal{N}(\mu_*, \mathbf{\Sigma}_*)$. Los parámetros los podremos ver detallados en 2.6, donde hemos incluido en la notación: $\mathbf{K_{xx}} = k(\mathbf{X}, \mathbf{X})$, $\mathbf{K_{xx_*}} = k(\mathbf{X}, \mathbf{X}_*)$, $\mathbf{K_{x_*x}} = k(\mathbf{X}_*, \mathbf{X})$ y $\mathbf{K_{x_*x_*}} = k(\mathbf{X}_*, \mathbf{X}_*)$.

$$\begin{aligned} \mu_* &= \mathbf{K_{x_*x}}(\mathbf{K_{xx}^{-1}} + \sigma_n^2 I)^{-1}\mathbf{y} \\ \Sigma_* &= \mathbf{K_{x_*x_*}} - \mathbf{K_{xx_*}}(\sigma_n^2 I + \mathbf{K_{xx}})^{-1}\mathbf{K_{x_*x}} \end{aligned} \tag{2.6}$$

Recordemos que todas estas operaciones las podemos hacer en forma cerrada, por lo que al final nuestro resultado sigue siendo una distribución gaussiana (sobre funciones) gobernada por un vector de medias y matriz de covarianzas.

2.1.1. Funciones de Kernel.

Una vez definido la parte esencial del proceso gaussiano, podemos centrarnos en detalles algo más específicos del mismo. En concreto, y por su importancia, en la función de kernel. Esta función, se encarga de medir la similitud, conforme al criterio elegido, entre pares de puntos. Al final, esto nos permite construir una matriz con la similitud entre todos los puntos conocidos [16]:

$$\mathbf{K} = \begin{pmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ \vdots & \vdots & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{pmatrix}$$

Esta matriz, conocida como matriz de Gram [16], debe contar con dos propiedades que en nuestro caso resultan imprescindibles. La primera es que la similitud entre x_i y x_j debe ser la misma que entre x_j y x_i. Cuando se compute la función de kernel de un conjunto de puntos consigo mismo obtendremos, por lo tanto, una matriz cuadrada y simétrica con respecto a su diagonal [16].

La segunda es que dicha matriz sea definida positiva [16], esto es para garantizar que sea invertible, una necesidad surge de la ecuación 2.6. Para garantizar que estas propiedades se cumplan, es importante escoger una función de kernel adecuada. Dado que en nuestro proceso estamos asumiendo que nuestras variables aleatorias tienen una distribución normal, lo más adecuado será utilizar un kernel gaussiano. En nuestro caso, utilizaremos el denominado *square exponential kernel* o SE, cuya función podemos ver en la ecuación 2.7. Atendiendo a dicha ecuación, se puede observar que esta depende de una serie de valores conocidos como hiperparámetros, estos son: el length scale característico (l) que nos servirá para *suavizar* nuestra función; el amplificador de la varianza muestral (σ_f^2), utilizado para ajustar la amplitud de nuestra función; y finalmente, el ruido gaussiano (σ_n^2), que como su propio nombre indica, nos permitirá modelar el ruido de nuestras observaciones [6]. En la figura 2.2

se muestra el impacto de las diferentes configuraciones en nuestro modelo.

$$k(x, x') = \sigma^2 exp(-\frac{(x - x')^2}{2l^2}) \tag{2.7}$$

Si por el contrario, en los datos existiesen otro tipo de relaciones, por ejemplo, una que se repite a lo largo del tiempo, podríamos utilizar un kernel con distintas propiedades, por ejemplo un kernel periódico. O incluso una combinación entre estos para trasladar a nuestro kernel gaussiano la relación periódica de forma local. Solo hay que fijarse en la ecuación 2.6, y ver la fuerte dependencia que existe entre los parámetros del proceso y el kernel, para darse cuenta de lo vital que resulta a la hora de modelar nuestro problema.

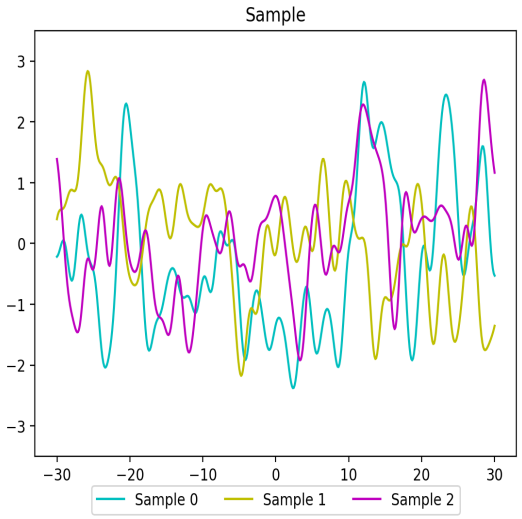
2.1.2. Ajustando el modelo a los datos

Ahora que conocemos el impacto de las funciones de kernel y de la correcta configuración de sus hiperparámetros, resulta apropiado indagar en los métodos que tenemos para escoger la configuración que maximice nuestro ajuste. Una de las técnicas que se usan es la maximización de la verosimilitud marginal [17]. Para hacer más sencilla esta tarea, es bastante común tomar el logaritmo de nuestra expresión (ecuación 2.5) obteniendo 2.8 [6].

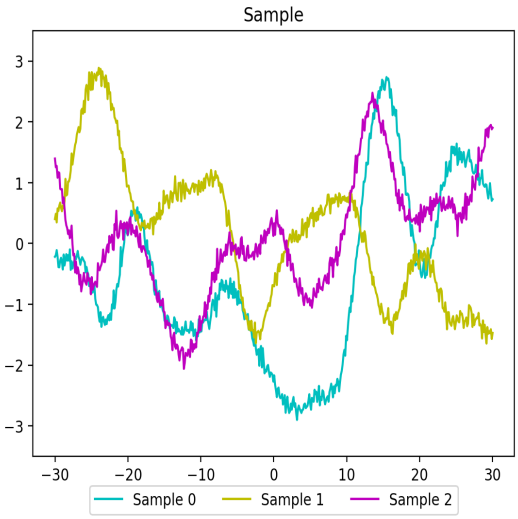
$$\log(p(\mathbf{y}|\mathbf{X}) = -\frac{1}{2}\mathbf{y}^T(K^{-1} + \sigma_n^2 I)\mathbf{y}\frac{1}{2}\log|K + \sigma_n^2 I| - \frac{N}{2}\log(2\pi) \tag{2.8}$$

Esto es debido, principalmente, a que nuestra función es suave y derivable, por lo que la operativa para optimizar no se vuelve muy compleja. Además, dado que estamos trabajando con expresiones exponenciales (gaussianas) tomar el logaritmo hace los cálculos más fáciles y, en computación, puede prevenir el *underflow* [5]. Por otro lado, si bien es cierto que no estamos trabajando con la expresión original, dado que la transformación de nuestra verosimilitud se realiza mediante una función monótona creciente, podemos conservar los máximos y mínimos. Sirve por lo tanto a nuestro propósito para derivar con respecto a al conjunto de hiperparámetros $\theta = (l, \sigma_n^2, \sigma_f^2)$.

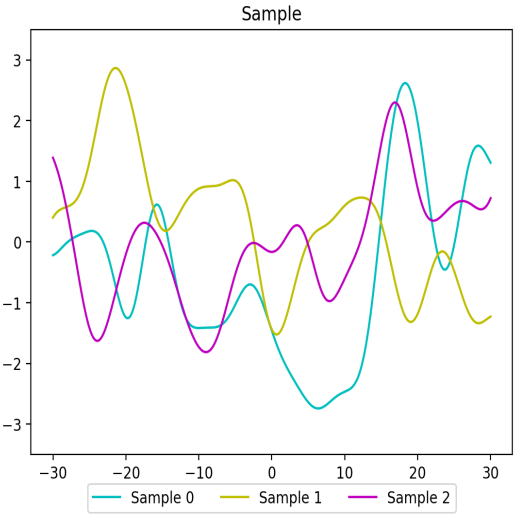
Dado que la resolución directa en espacios paramétricos de dimensión elevada (lenght scale) puede resultar muy engorrosa, el método que vamos a utilizar para maximizar la función objetivo 2.8, es el de ascenso por gradiente -o descenso si invertimos el signo-. Este método nos acercará gradualmente a nuestro máximo, pero no estará exento de problemas, y como veremos en la próxima sección, el almacenamiento de la matriz K de dimensiones $N \times N$ escalara cuadráticamente, y la inversión de la misma tendrá coste computacional cúbico con respecto al número de datos de entrada.



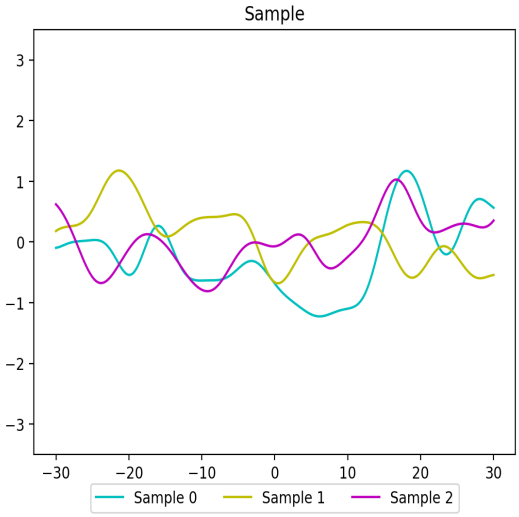
(a) $(l,\sigma_n^2,\sigma_f^2) = (1,\ 1e\text{-}10,\ 1)$.



(b) $(l,\sigma_n^2,\sigma_f^2) = (3,\ 0.01,\ 1)$.



(c) $(l,\sigma_n^2,\sigma_f^2) = (3,\ 1e\text{-}10,\ 1)$.



(d) $(l,\sigma_n^2,\sigma_f^2) = (3,\ 1e\text{-}10,\ 0.1)$.

Figura 2.2: En estas subfiguras se presentan funciones muestreadas a partir de un proceso gaussiano con diferentes configuraciones de los hiperparámetros. En la subfigura 2.2(a), inicializamos los hiperparámetros a unas cantidades comunes dentro de este tipo de problemas. En la 2.2(b) incrementamos el nivel de ruido y el *lenght scale*. En la subfigura 2.2(c), incrementamos exclusivamente el *lenght scale* sobre la 2.2(a) y en la subfigura 2.2(d) reducimos la varianza sobre la anterior. Todas estas imágenes han sido generadas utilizando el test 1 del apéndice B.1.6 mediante el código del apéndice B.6 (con el fichero de configuracion default_conf.json y –n-size 500) usando distintas configuraciones para los hiperparámetros ($-$l,$-$n y $-$v).

2.2. Procesos Gaussianos con Big Data.

Hasta este momento, hemos presentado el proceso gaussiano como una solución flexible, que es capaz de modelar un problema complejo utilizando, exclusivamente, las propiedades de la distribución gaussiana. Esto nos permite interpolar los datos, y ofrecer predicciones al mismo tiempo que modelamos la incertidumbre [6] [5].

No obstante, no todo son bondades. En particular nos encontramos con que invertir la matriz de Gram de dimensión N, $\mathbf{K}$, tiene un coste computacional cúbico ($O(N^3)$) y cuadrático ($O(N^2)$) con respecto al consumo de memoria. Esto hace impracticable el computo del proceso para grandes conjuntos de datos.

Por fortuna, existen diferentes técnicas que nos permiten evadir este obstáculo. En los sucesivos apartados expondremos un método que realiza inferencia variacional estocástica. Como novedad, incorporaremos un conjunto adicional de variables latentes con las que pretendemos aproximar el posterior real de nuestro proceso sin incurrir en el mismo coste [7]. Diferentes autores [7] [2] [3] se refieren a este conjunto de variables latentes como $\mathbf{u}$ y al conjunto de entrada $\overline{\mathbf{X}}$ se le denomina *inducing set* y su relación viene dada por $\mathbf{u} = f(\overline{\mathbf{X}})$. Este planteamiento sigue las líneas generales expuestas en la sección 2.1, en este caso $\overline{\mathbf{X}}$ sería el conjunto adicional -de tamaño M- de los vectores de características.

La introducción de las variables latentes $\mathbf{u}$ nos va a dejar con un posterior ampliado que podemos descomponer como en la ecuación 2.9. En este punto, todavía no hemos hecho un gran cambio, y de hecho, podemos recuperar el posterior original marginalizando $\mathbf{u}$ [2].

$$p(\mathbf{f}_*, \mathbf{f}, \mathbf{u}|\mathbf{y}) = p(\mathbf{f}_*, \mathbf{f}|\mathbf{u})p(\mathbf{u}|\mathbf{y}) \tag{2.9}$$

En nuestro camino para conseguir la reducción de costes, vamos a asumir la independencia de $\mathbf{f}_*$ y $\mathbf{f}$ dado $\mathbf{u}$, y a tratar de aproximar nuestra distribución posterior mediante el uso de una distribución variacional [2]. En concreto, tomaremos $q(\mathbf{u}) \sim \mathcal{N}(\mathbf{u}|\mathbf{m}, \mathbf{S})$ para aproximar $p(\mathbf{u}|\mathbf{y})$ y con ello nuestro posterior, esto dará lugar a la ecuación 2.10 [2], donde $q(\mathbf{f}_*, \mathbf{f}, \mathbf{u}) \simeq p(\mathbf{f}_*, \mathbf{f}, \mathbf{u}|\mathbf{y})$. Por supuesto, deberemos elegir los parámetros $\mathbf{m}$ y $\mathbf{S}$, así como los puntos inductores, que maximicen la similitud entre estas distribuciones; aunque este proceso se detallará en siguientes apartados.

$$q(\mathbf{f}_*, \mathbf{f}, \mathbf{u}) = p(\mathbf{f}_*|\mathbf{u})p(\mathbf{f}|\mathbf{u})q(\mathbf{u}) \tag{2.10}$$

Si de nuestra distribución posterior aproximada marginalizamos $\mathbf{f}$ y $\mathbf{u}$ obtendremos la distribución predictiva aproximada $\mathbf{f}_* \sim \mathcal{N}(\mu_*^{\mathbf{q}}, \mathbf{\Sigma}_*^{\mathbf{q}})$, cuyos parámetros vienen definidos en 2.11. Al igual que en la ecuación 2.6, volveremos a utilizar subíndices para denotar sobre que conjuntos se construye la matriz $\mathbf{K}$.

$$\mu_*^q = \mathbf{K_{x_*u}K_{uu}^{-1}u}$$
$$\Sigma_*^q = \mathbf{K_{x_*x_*}} - \mathbf{K_{x_*u}K_{uu}^{-1}K_{ux_*}} + \mathbf{K_{x_*u}}B\mathbf{K_{ux_*}} \tag{2.11}$$
$$B = (\mathbf{K_{uu}} + \sigma^{-2}\mathbf{K_{ux}K_{xu}})^{-1}$$

En las próximas secciones, veremos las técnicas utilizadas para seleccionar los parámetros de nuestra distribución $q(\mathbf{u})$. Sin embargo, estos no nos hacen falta para observar como la introducción del nuevo conjunto de variables afecta al tamaño de la matriz $\mathbf{K}$. Para valores de M pequeños, podremos realizar predicciones con costes más aceptables que cuando utilizamos la entrada completa (N). En particular, estaremos hablando de costes computacionales de $O(MN^2)$, mientras que para el almacenamiento necesitaremos $O(MN)$ [2].

2.2.1. Inferencia Variacional

La inferencia variacional, trata, básicamente, de tomar un funcional [4] o distribución variacional, de forma que, a la hora de optimizar, podamos explorar un conjunto de diferentes modelos para aproximar nuestra distribución posterior aproximada $q(\mathbf{f}, \mathbf{u})$ a la real $p(\mathbf{f}, \mathbf{u}|\mathbf{y})$ [2]. Dado que computar el proceso de optimización sobre todas las posibles funciones y sus parámetros es bastante costoso, podemos restringir la familia de distribuciones puede tomar nuestro funcional [5]. En el caso que nos atañe, la aproximación se hace mediante la ecuación 2.10 con $p(\mathbf{f}|\mathbf{u})$ fijo y $q(\mathbf{u})$ ajustable y restringido a la familia exponencial (gaussiana) [2]. Al final, esta decisión tan sencilla, mejora nuestro coste computacional y nos permite centrarnos en la búsqueda de los parámetros de la distribución mediante técnicas que veremos en el apartado 2.2.2. Otra de las virtudes de este proceso será que nos permitirá seleccionar nuestro *inducing set* de forma conjunta con los hiperparámetros del kernel [5] [2].

2.2.2. Evicence Lower Bound, EM y la Divergencia de Kullback-Leibler

El algoritmo de la maximización de la esperanza -expectation maximization o EM en inglés-, es el método que vamos a utilizar para determinar la calidad de la aproximación que hacemos mediante inferencia variacional. Para empezar a entender este algoritmo, tenemos que fijarnos primero, en los cambios que experimenta la verosimilitud marginal 2.8 tras introducir la nueva distribución a aproximar. En concreto, tenemos la ecuación 2.12, que podemos obtener aplicando la desigualdad de Jensen o la regla del producto [5] [3].

$$\log p(\mathbf{y}) = \mathcal{L}(q) + KL(q(\mathbf{f}, \mathbf{u})||p(\mathbf{f}, \mathbf{u}|\mathbf{y})) \tag{2.12}$$

El primer término -$\mathcal{L}$- es una cota inferior de la verosimilitud marginal [5] conocida como *evidence lower bound* o *ELBO* [18]. Su expresión viene definida en la ecuación 2.13.

[4] Función que toma funciones en su argumento.

$$\begin{aligned}\mathcal{L}(q) &= \mathrm{E}_{q(\mathbf{f},\mathbf{u})}\left[\log \frac{p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{u})p(\mathbf{u})}{q(\mathbf{f},\mathbf{u})}\right] = \mathrm{E}_{q}(\mathbf{f},\mathbf{u})\left[\log \frac{p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{u})p(\mathbf{u})}{p(\mathbf{f}|\mathbf{u})q(\mathbf{u})}\right] \\ &= \mathrm{E}_{q(\mathbf{f},\mathbf{u})}\left[\log \frac{p(\mathbf{y}|\mathbf{f})p(\mathbf{u})}{q(\mathbf{u})}\right] = \mathrm{E}_{q(\mathbf{f})}\left[\log p(\mathbf{y}|\mathbf{f})\right] - KL(q(\mathbf{u})|p(\mathbf{u}))\end{aligned} \tag{2.13}$$

En cuanto al segundo término -KL-, se le conoce como la divergencia de Kullback-Leibler o entropía relativa (ecuación 2.14), y se encarga de medir la cantidad de información adicional media que aporta una variable aleatoria sobre otra; es decir, nos permite conocer como de similares son la distribución original $p(\mathbf{f},\mathbf{u}|\mathbf{y})$ con respecto a la aproximada $q(\mathbf{f},\mathbf{u})$. Dado que la divergencia de Kullback-Leibler es una cantidad relativa y no simétrica (ecuación 2.14), siempre tendremos que $KL(q||p) \geq 0$, dado que no existe dependencia entre $\log p(\mathbf{y})$ y $q(\mathbf{u})$ alcanzaremos el valor máximo cuando $q(\mathbf{f},\mathbf{u}) = p(\mathbf{f},\mathbf{u}|\mathbf{y})$ [5] y la divergencia KL desaparezca [6] [5]. Cuando no conozcamos nuestra distribución original, será suficiente con aproximarla mediante un conjunto finito de datos muestreados de dicha distribución [5].

$$\mathrm{KL}(p||q) = -\int p(x)\ln q(x)dx - \left(\int p(x)\ln p(x)dx\right) = -\int p(x)\ln\left\{\frac{q(x)}{p(x)}\right\}dx \tag{2.14}$$

Ahora que ya tenemos nuestra verosimilitud marginal es el momento de aplicar el algoritmo E.M. En general, este se aplicará de forma iterativa en dos pasos. En el primero, maximizamos la cota inferior $\mathcal{L}(q)$ con respecto a $q(\mathbf{u})$ pero manteniendo fijados parámetros integrados. En el segundo actualizamos los hiperparámetros según los resultados obtenidos en el primer paso. En este momento, y si no hemos alcanzado ya el máximo, podremos observar como la cota inferior aumenta su valor y el de la función de likelihood en consonancia. Esto es debido a que nuestro lower bound estaba maximizado utilizando los parámetros anteriores, y al utilizar los nuevos la divergencia de KL vuelve a aumentar. Este proceso se repetirá hasta que KL se aproxime a 0 y la verosimilitud marginal deje de aumentar, en este punto la cota inferior alcanzará el máximo ajuste para los parámetros dados.

2.2.3. Inferencia variacional estocástica

Hasta este momento, lo que hemos visto, basado en el método de propuesto por Michelis K.Titsias [2], nos va a servir para obtener nuestro ELBO, y con el reducir el coste computacional a $O(MN^2)$ y a $O(MN)$ el de almacenamiento. Esto, por supuesto, para una entrada de tamaño N y un inducing set de tamaño M, donde buscamos $N >> M$. No obstante, siguen existiendo conjuntos de datos que, por su tamaño, es complicado que nuestro hardware pueda procesar. Es aquí donde podemos utilizar las soluciones propuestas por [3] y [19], en lo que llaman inferencia variacional estocástica.

El proceso es sencillo, simplemente hay que segmentar la entrada $\mathbf{X}$ en subconjuntos y procesarlos a medida que avanzamos en la dirección del gradiente natural [3], esto nos permitirá conseguir

[5] Esto sería lo ideal, pero es complicado cuando se plantean restricciones como la de la familia de distribuciones, por esta razón nos conformaremos con soluciones aproximadas [5].

[6] Por esta razón, la maximización del ELBO se considera equivalente a la minimización de la divergencia KL [5].

buenas soluciones antes incluso de recorrer todas las instancias de datos, rompiendo en la práctica la dependencia del coste con N. Por esta razón, y a diferencia de la cota mínima propuesta por Titsias [2], necesitamos conservar, explícitamente, nuestra distribución variacional $q(\mathbf{u}) = \mathcal{N}(\mathbf{u}|\mathbf{m_q}, \mathbf{S})$ como en la ecuación 2.15 [3].

$$\mathcal{L} = \sum_{i=1}^{N} \{E_q(f_i) \log p(y_i|f_i)\} - \mathbf{KL}(q(\mathbf{u})||p(\mathbf{u})) \tag{2.15}$$

Desarrollando el primer término de 2.15 obtenemos 2.16, donde usamos μ_{f_i} como la media de f_i y k_{f_i} como la varianza de la misma.

$$\begin{aligned} E_{q(f_i)} \log p(y_i|f_i) &= E_{q(f_i)} \log \mathcal{N}(y_i|f_i, \sigma_n^2) \\ &= -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(y_i^2 - 2\mu_{f_i} y_i + \mu_{f_i}^2 + k_{f_i}) \end{aligned} \tag{2.16}$$

Derivando la expresión completa con respecto a los parámetros de nuestra distribución variacional conseguiremos las ecuaciones que podemos utilizar para avanzar en con pasos de tamaño α en la dirección de nuestro gradiente natural [3]. Esto lo haremos aproximando 2.15 (y por lo tanto su gradiente) mediante minibatches con la forma 2.17, donde $|B| \simeq M$.

$$\mathcal{L} \approx \frac{N}{|B|} \sum_{i}^{B} E_q(f_i) \log p(y_i|f_i)) - \mathbf{KL}(q(\mathbf{u})||p(\mathbf{u})) \tag{2.17}$$

En la práctica usaremos el algoritmo Adam [20], para avanzar en la dirección de nuestro máximo. No obstante, este es un algoritmo de minimización, por lo que será importante invertir primero el signo. De esta forma podremos optimizar la distribución q -seleccionando los puntos inductores $\overline{\mathbf{X}}$ más adecuados- y los hiperparámetros -l,σ_f^2 y σ_n^2- de forma conjunta.

3

Diseño

3.1. Análisis de requisitos.

Los requisitos funcionales que hemos planteado para desarrollar nuestro proyecto son los siguientes:

RF-1.– Importar datos en csv.

RF-2.– Autogenerar datos para hacer pruebas.

RF-3.– Particionar los datos explícitamente en conjuntos de entrenamiento y test.

RF-4.– Particionar los datos aleatoriamente en conjuntos de entrenamiento y test.

RF-5.– Dadas las particiones de entrenamiento y test, segmentarlas para realizar la inferencia en minibatches.

RF-6.– Realizar inferencia utilizando procesos gaussianos.

RF-7.– Optimizar hiperparámetros de los procesos gaussianos mediante las técnicas expuestas en 2.1..

RF-8.– Realizar inferencia sobre los procesos gaussianos aproximados.

RF-9.– Optimizar hiperparámetros de los procesos gaussianos aproximados mediante las técnicas expuestas en 2.2.3.

RF-10.– Implementar la función de kernel Radial Basis Function (RBF)

RF-11.– Implementar las restricciones para los hiperparámetros.

RF-12.– Implementar un módulo que permita calcular como de bueno es nuestro ajuste con diferentes criterios: Error Cuadrático Medio (MSE), Log likelihood.

RF-13.– Medir los tiempos de entrenamiento y de test.

RF-14.– Tener la capacidad para poder variar la configuración (GPU, número de batches...) de nuestro código desde la entrada. Esto podrá hacerse desde ficheros de configuración o desde la línea de comandos. La especificación completa de opciones puede verse en el apéndice D

Los no funcionales:

RNF-1.– Usabilidad. Es necesario contar con un sistema de *feedback* que permita al usuario conocer las consecuencias de su interacción con el programa.

RNF-2.– Mantenibilidad. Es necesario contar con un sistema que permita analizar de forma sencilla que está haciendo la aplicación en todo momento. De esta forma podremos facilitar la depuración, detectando bloqueos y errores que pudieran alterar la correcta ejecución del programa.

3.2. Análisis de la funcionalidad

Todos los módulos y ficheros de nuestro proyecto los podemos agrupar prácticamente en las siguientes categorías o funcionalidades:

F-1.– Lanzar el programa: básicamente gestionan la configuración y el flujo del programa.

F-2.– Inferencia: aquí distinguimos los módulos dedicados a realizar inferencia con procesos gaussianos *GaussianProcess* y *SparseGaussianProcess*. Se define una interfaz que permite normalizar los datos, ajustar el modelo (train), predecir (test) y optimizar los hiperparámetros. Por supuesto, existe alguna funcionalidad adicional como las encargadas de escalar los datos, o simplemente funciones auxiliares para realizar alguna de las tareas previamente comentadas.

F-3.– Funciones de covarianza: estos módulos se encargaran de definir las diferentes funciones de kernel y de generar su matriz de Gram. A su vez, se encargarán de almacenar los hiperparámetros y de prepararlos, en su conjunto, para ser optimizados. La funcionalidad ampliada la detallaremos en el apartado 3.3.2.

F-4.– Importar datos: en esta categoría encontramos los módulos *Dataset* y *Partition*. Se encargan de importar los datos de un fichero en formato csv o autogenerarlos, y particionarlos en conjuntos de entrenamiento y test. Aquí también encontraremos la funcionalidad para generar batches de particiones, y seleccionar datos para conjuntos aproximados. En la sección 3.3.3 veremos que estrategias hemos implementado para la selección de datos.

F-5.– Operaciones auxiliares: leer la configuración del programa, medida de rendimientos, operaciones con matrices y el registro (y en ocasiones cálculo) de métricas de la calidad del ajuste (MSE, elbo, etc).

3.3. Modelado de los requisitos.

En las figuras 3.1 y 3.2 presentamos unos diagramas -de clase y de secuencia- generales, esto quiere decir que algunos de los componentes pueden estar simplificados con el objetivo de hacer énfasis en las relaciones, funciones y variables que realmente importan granulando la información para mejorar la comprensión. Es más, módulos como los que están integrados en el *Lanzador* no están configurados como clases, pero a efectos de diseño es útil tratarlos así para modelar las relaciones con el resto de código. Por supuesto, aunque algunos de estos detalles omitidos tienen menor relevancia, pueden seguir siendo necesarios y por ello los detallaremos en las sucesivas secciones.

3.3.1. Inferencia utilizando procesos gaussianos

El objetivo final de este trabajo es verificar la reducción de costes al realizar inferencia mediante procesos gaussianos. En nuestro caso, contemplaremos los siguientes escenarios: procesos gaussianos, procesos gaussianos aproximados y utilización de GPU para computo. Por esta razón, cobra importancia diseñar una interfaz lo suficientemente flexible como para abarcar los diferentes casos de uso derivados sin complicarnos demasiado. En la figura 3.1, ya se modela con suficiente precisión la parte del proceso gaussiano y su versión aproximada. Por otro lado, el uso de la GPU está abstraído

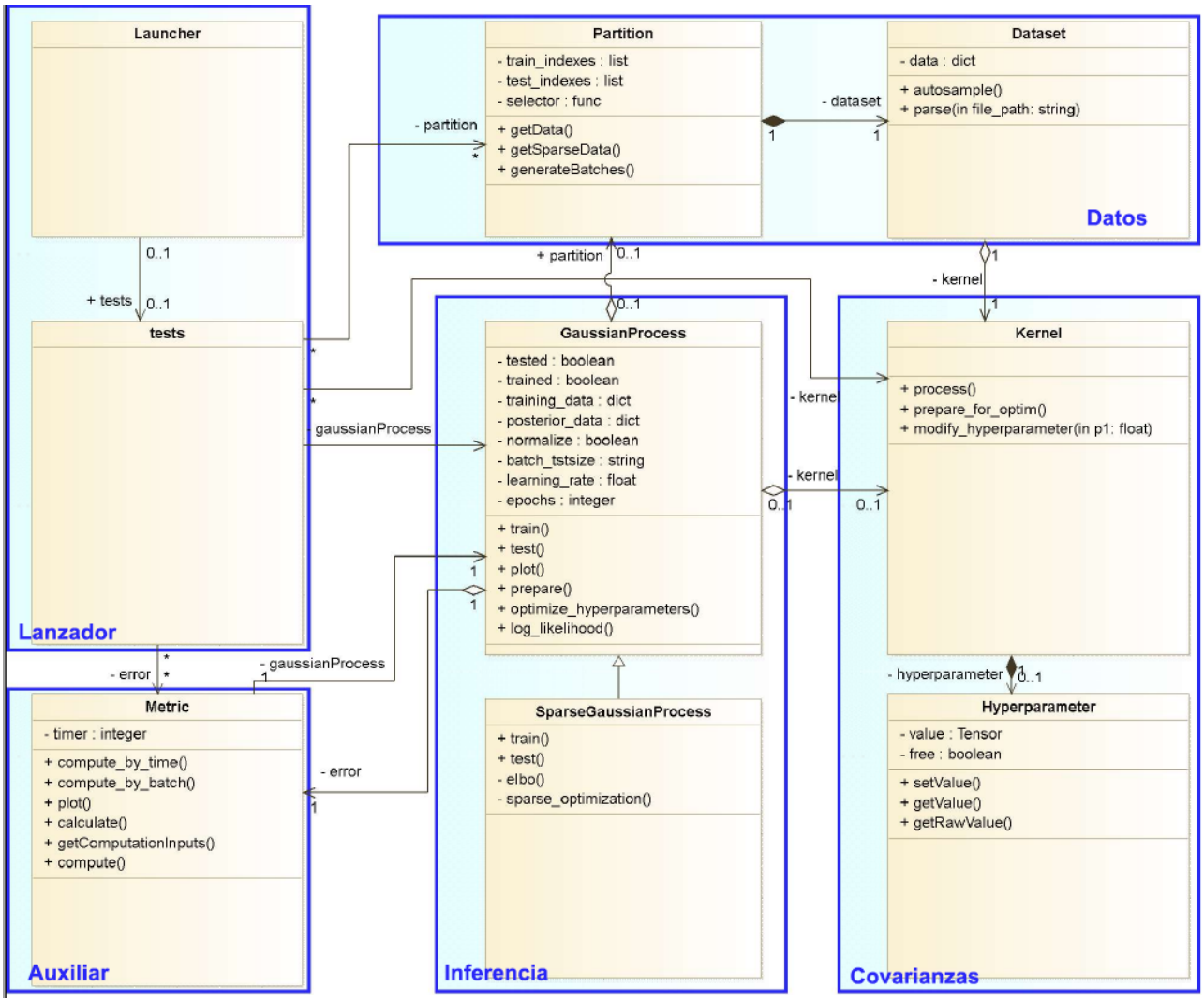


Figura 3.1: En esta figura se presenta el diagrama de clases general del proyecto. En azul se han resaltado las funcionalidades introducidas en la sección 3.2. Además, los detalles de modelado de la clase *Partition*, *Metric* y *Kernel* se han omitido para ganar claridad y se modelaran por separado en sus respectivos apartados.

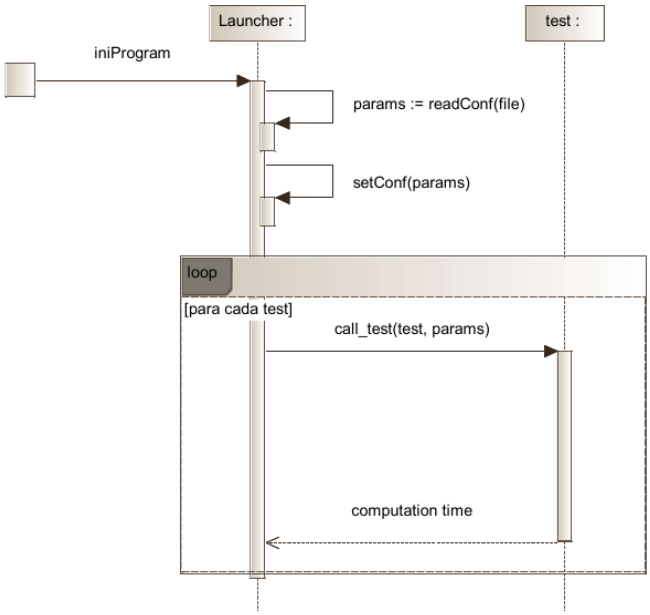


Figura 3.2: En esta figura, podemos ver de forma muy esquemática como se pretende ejecutar el código. Hay que tener en cuenta que la funcionalidad invocada desde las 'clase' *Launcher* y *tests*, no se corresponde exactamente con métodos de clase. En concreto, readConf(), es una función auxiliar del modulo input y las otras dos se corresponden con código que se ha agrupado para ver la secuencia de funcionalidad.

en la tecnología subyacente que vamos a utilizar, de ello hablamos en la sección 4.1. De esta forma, solo nos queda especificar como va a ser la interacción entre los componentes de nuestro programa para realizar la inferencia. En la figura 3.3 podemos ver dicha interacción para proceso gaussiano aproximado, el modelado también sirve para uno normal.

3.3.2. Funciones de Kernel

Las funciones de kernel son uno de los pilares sobre los que se apoyan los procesos gaussianos. Aquí, concretamente, se utilizará la función RBF. Sin embargo, como ya se comentó en el apartado 2.1.2, estamos ante un entorno rico y flexible que permite utilizar diferentes funciones de kernel, incluidas las combinaciones entre ellas. A esto se le suma la necesidad de restringir los valores que tienen nuestros hiperparámetros para garantizar la corrección de la matriz de Gram. Esta funcionalidad la abstraeremos mediante la clase *Hyperparameter*. Por otro lado, teniendo estos factores en cuenta para el diseño, podemos hacer más sencillo un hipotético trabajo futuro. El diseño de la funcionalidad de kernel lo podemos encontrar más detallado en la figura 3.4.

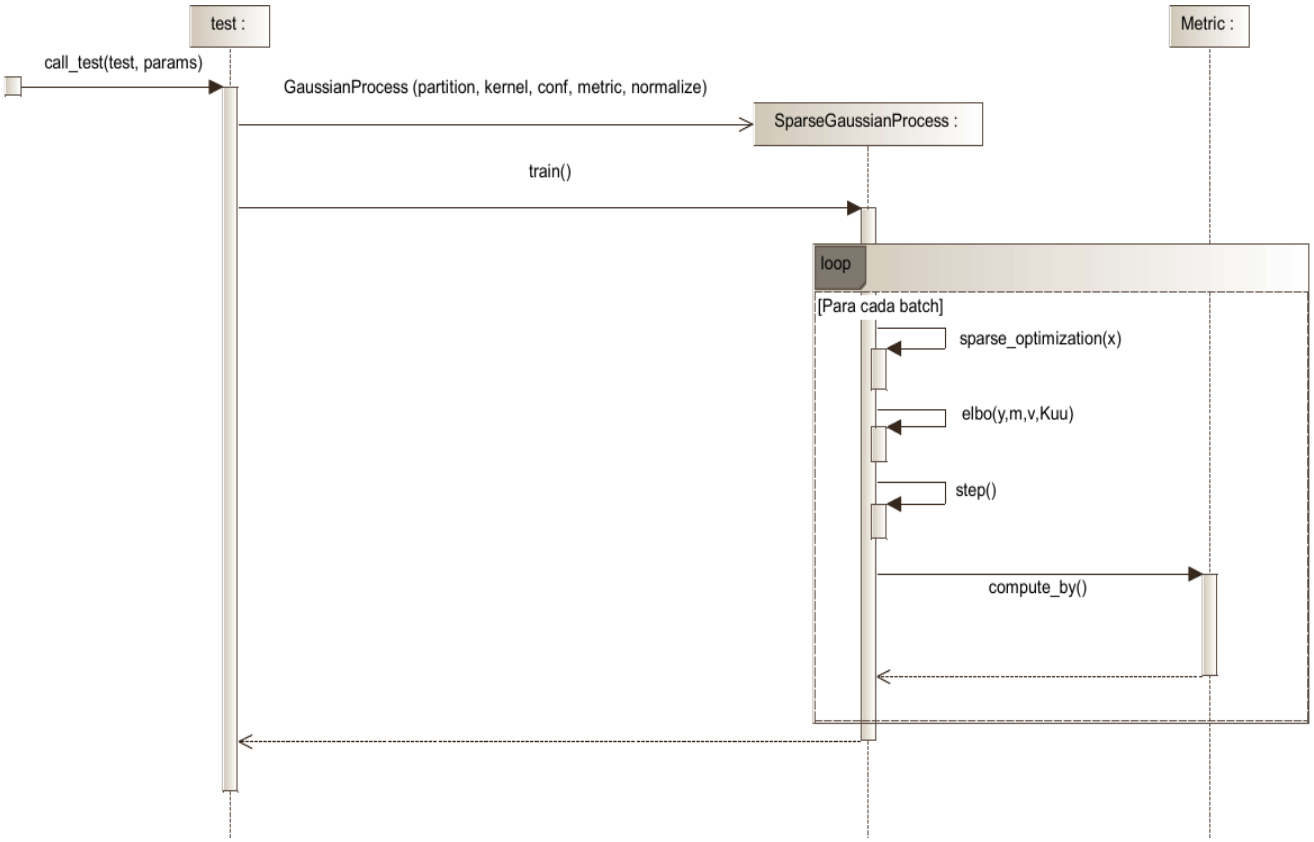
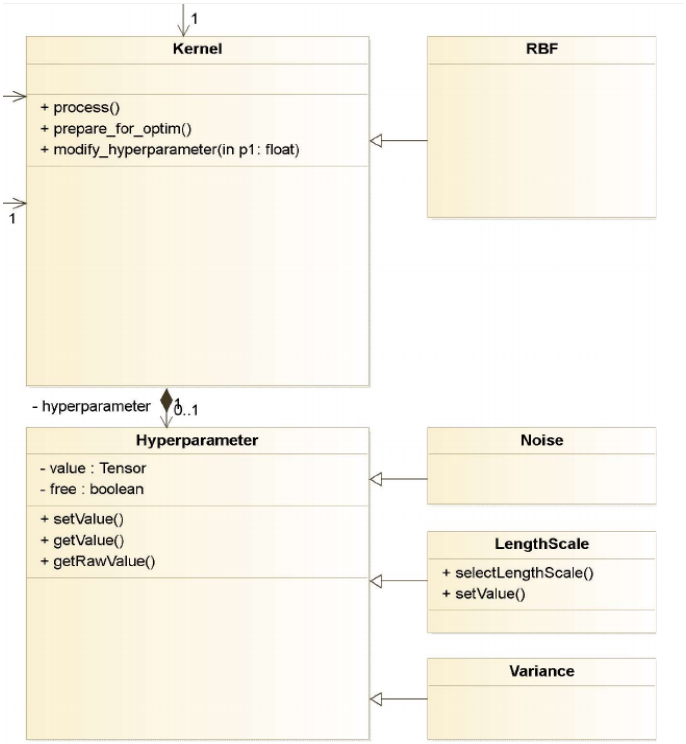
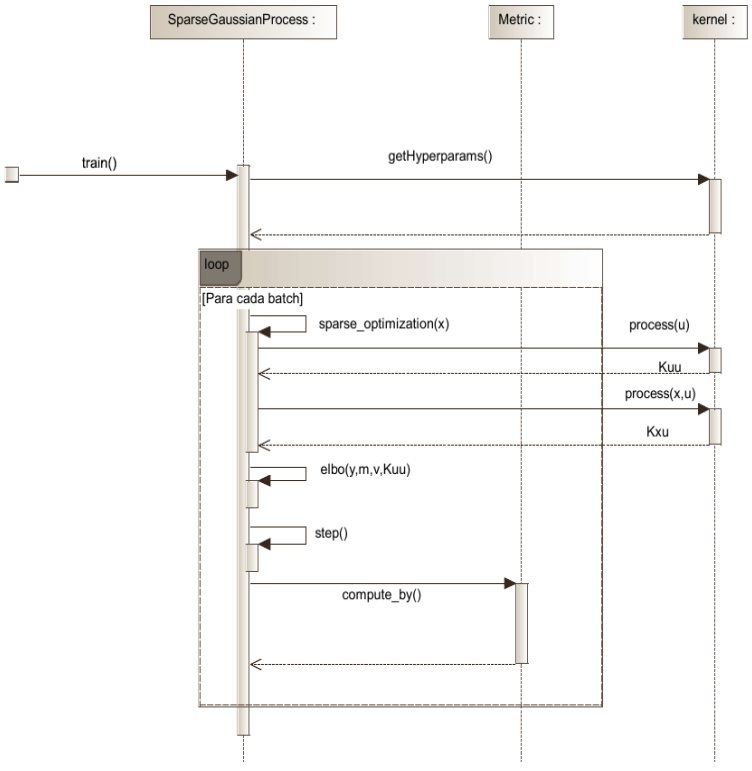


Figura 3.3: Diagrama de secuencia para realizar inferencia con un proceso gaussiano aproximado. Antes de crear el proceso gaussiano habrá que crear los diferentes objetos que aparecen en su constructor.



(a) Diagrama de clases para la funcionalidad de kernel.



(b) Diagrama de secuencia para el entrenamiento del GP.

Figura 3.4: En la subfigura 3.4(a) podemos ver el diagrama de clases que incorpora los detalles omitidos en la figura 3.1, y se encarga de modelar las diferentes funciones de kernel y abstracción de los hiperparámetros. Por otro lado, en la subfigura 3.4(b), se muestra la interacción genérica de la clase kernel con los módulos de inferencia durante el entrenamiento.

3.3.3. Carga y particionado de datos

La gestión de datos la vamos a extender con las clases *RandomPartition* y *ExplicitPartition*. Estas clases, que se pueden ver en la figura 3.5, nos van a aportar los criterios de selección para generar los conjuntos de train y test de la partición. Aquí se incluye la generación del conjunto de variables latentes para nuestro *SparseGaussianProcess*.

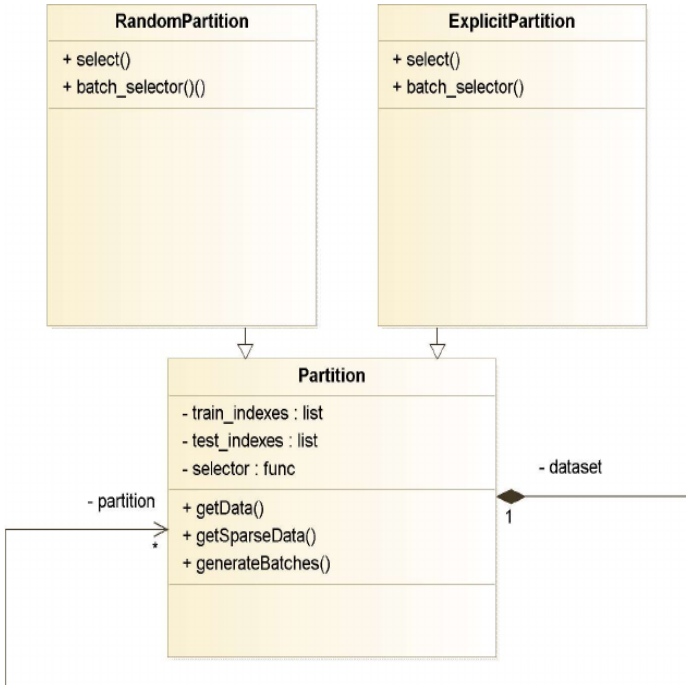


Figura 3.5: Diagrama de clases que modela los detalles y relaciones adicionales de la clase *Partition*.

En cuanto a la interacción con el resto de clases, es trivial, básicamente se crea un *Dataset* que carga los datos del fichero, en *Partition* se separan los datos en conjuntos train-test, y estos datos se recuperan por el proceso gaussiano en este formato o en batches si es un proceso gaussiano aproximado. Estos procesos se realizan en los constructores de los diferentes objetos, con la salvedad de la generación de batches que no se realiza hasta el instante antes de empezar a entrenar, esto se puede ver en la figura 3.6.

3.3.4. Modulos Auxiliares

En cuanto a los módulos auxiliares, están conformados por funciones auxiliares y alguna clase completamente estática cuyas funcionalidades se van a poder llamar desde el resto de clases. Por esta razón el enfoque que tomamos en esta subsección es textualmente más descriptivo y con un menor uso de diagramas.

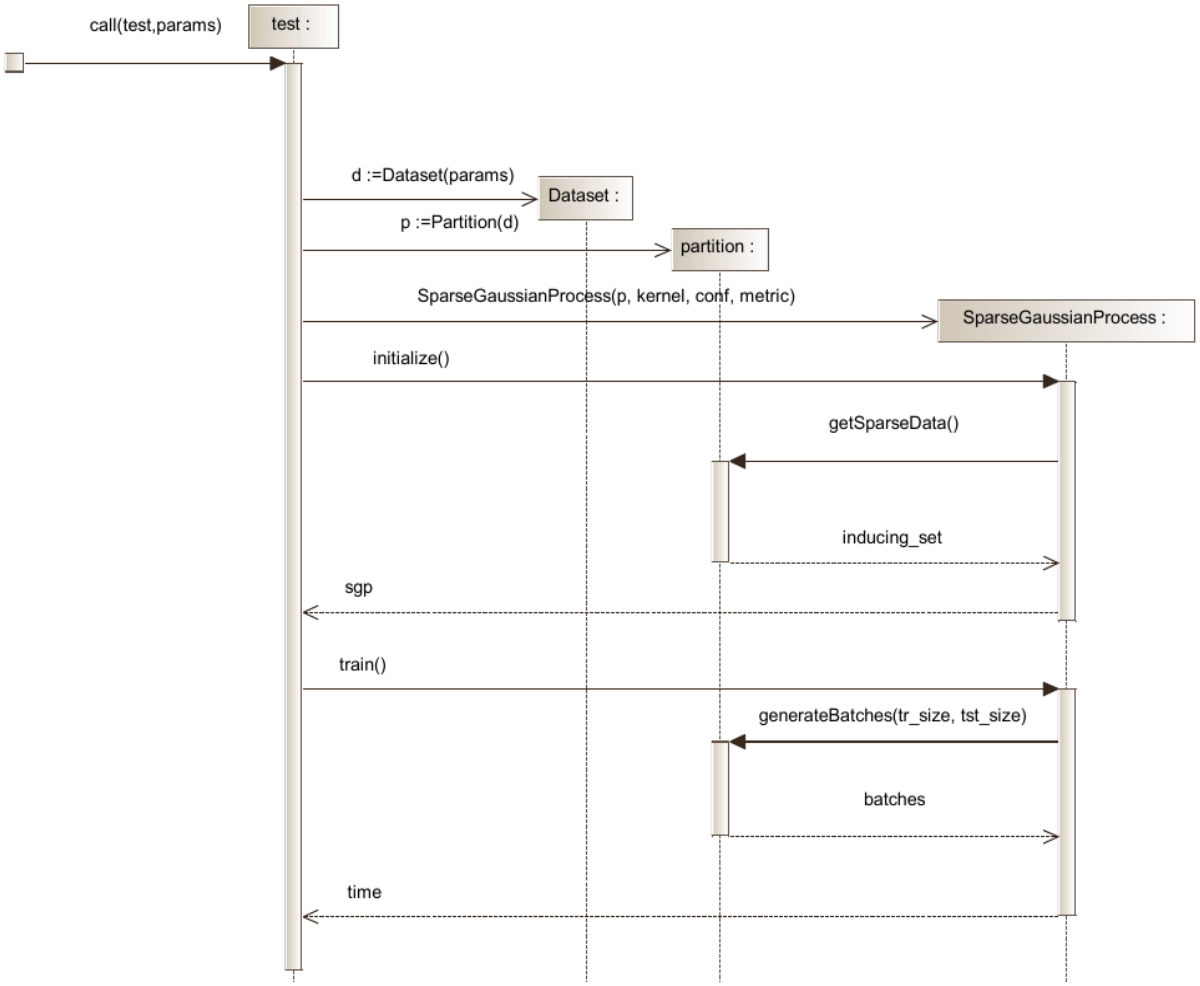


Figura 3.6: Diagrama de secuencia que modela la pila de llamadas durante la carga y el particionado de datos. Por simplicidad, cuando se modela la relación entre la partición y el train del SGP, se omiten las relaciones con el objeto *kernel* y *metric*; sin embargo en la practica estas estarían presentes después de la generación de batches y de acuerdo a lo detallado en la sección 3.3.2.

Módulo Input

El primer módulo auxiliar que vamos a ver es el de *input*. Este fichero aglutina la funcionalidad necesaria para leer el fichero con las opciones de configuración. Además, será posible sobrescribir cada una de estas opciones por línea de comandos, lo que nos va a ofrecer mucha flexibilidad a la hora de realizar las pruebas. Tanto las opciones del fichero de configuración como las de la línea de comandos las podemos ver en el apéndice D.

Módulo Logger

Diseñada como una clase completamente estática, tiene la funcionalidad necesaria para notificar al usuario 5 tipos de logs.

1.– Error: Cuando algún input es incorrecto, lo notifica y ejecuta la rutina de salida.

2.– Warning: Cuando en algún input o rutina faltan datos, pero se puede utilizar alguno por defecto o calcular a partir de otros.

3.– progress: 'barra de progreso' que indica el progreso sobre los algoritmos que se ejecutan dentro de bucles (épocas, batches).

4.– info: Notificaciones sobre el estado de la ejecución del programa.

5.– debug: logs para ver en tiempo real el valor de determinadas variables. Será útil para ver cómo evoluciona el elbo, los hiperparámetros o las métricas de calidad del ajuste.

Hay que tener en cuenta que tantos logs pueden aportar tanta información que, al final, no podemos encontrar la que necesitamos. Para solucionar este problema vamos a desarrollar unas opciones de configuración que permitan limitar qué tipos de log queremos ver y distinguirlos en nuestra pantalla. En el apéndice E se detalla su uso.

Módulo Metric

En este módulo se definen diversas métricas que nos pueden ayudar a determinar como de bien se ajusta nuestro modelo a los datos. Como podemos ver en la figura 3.7, nuestro diseño contempla una clase genérica *Metric*, de la que heredan los diferentes tipos de métricas, incluido un tipo especial de métrica -clase *Metrics*-, que consiste en un conjunto de métricas. Esta última es, probablemente, la que más utilidad nos va a aportar, ya que va a permitir configurar dinámicamente todas las métricas que se quieren calcular en un proceso sin necesidad de modificar el código del mismo, esta relación se puede ver en la figura 3.3.

Módulo performance

Este es un pequeño módulo funcional conformado por unas pocas funciones encargadas de tomar las mediciones de tiempos y de variar el tamaño de las entradas de nuestros algoritmos.

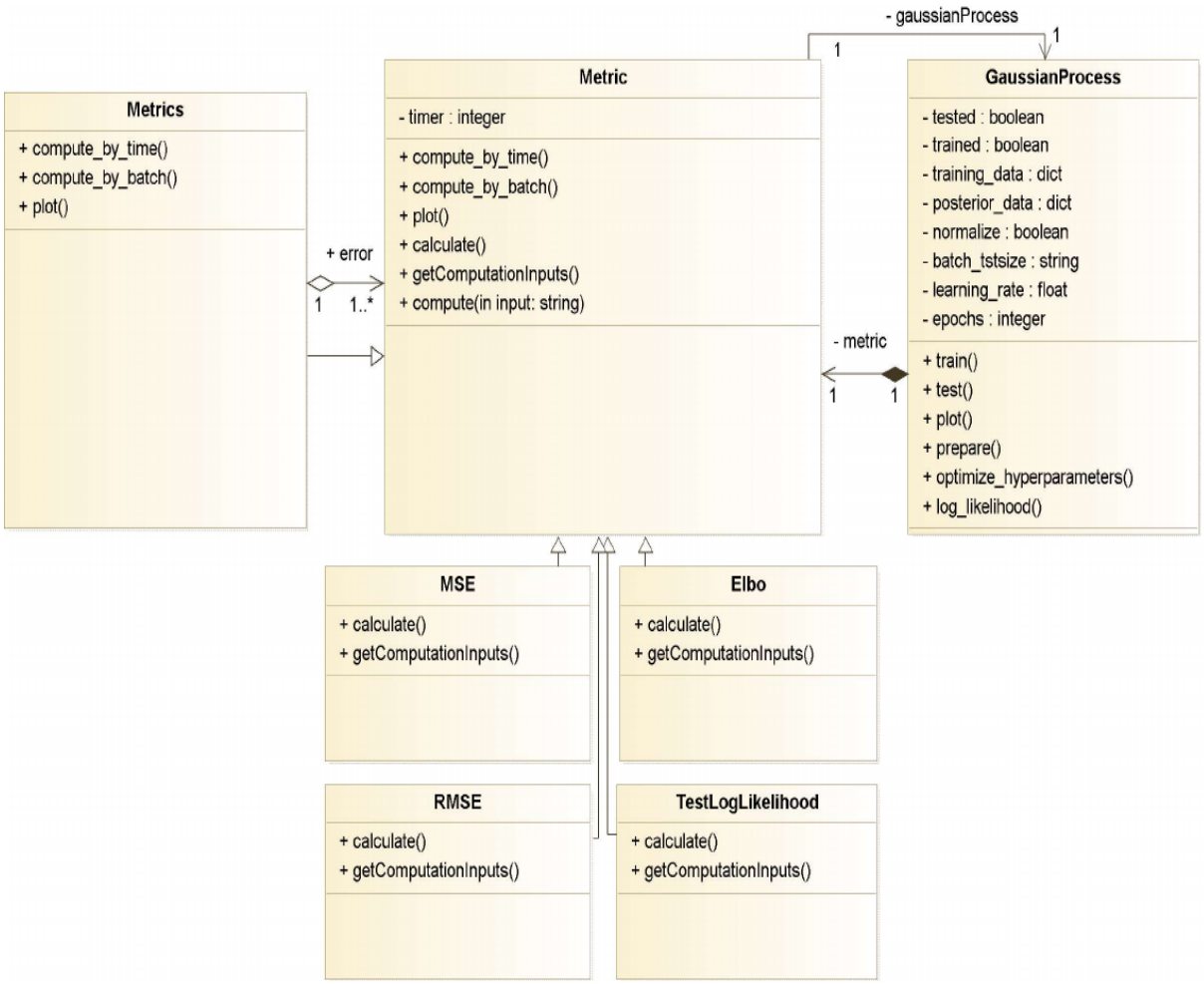


Figura 3.7: Diagrama de clases que extiende la clase *Metric*.

4

Desarrollo

En este capítulo vamos a concretar algunos detalles y decisiones que hemos tomado a la hora de implementar nuestro diseño. Para ilustrar estas cuestiones, haremos referencia a fragmentos concretos del código, el resto se podrá consultar en el apéndice B.6

4.1. Introducción a Pytorch.

Pytorch es una biblioteca -con API para python- enfocada principalmente al desarrollo de algoritmos de *deep learning* y para su ejecución sobre diferentes unidades de procesamiento (GPU y CPU) [21]. Sin embargo, en el camino hacia estos objetivos, se ha creado un ecosistema que aporta muchas facilidades para el desarrollo de otras técnicas de aprendizaje automático.

La principal ventaja que nos va a aportar es el soporte para los tensores. Este es un tipo de estructura que se encarga de guardar los datos, abstraer la unidad física en la que se ejecuta nuestro proceso y, por último, de guardar metainformación sobre las operaciones ejecutadas sobre dicho tensor. Es importante destacar que, a la hora de realizar el ajuste a los datos de nuestro modelo, dicha metainformación es crucial. La razón es que pytorch guarda esta información en forma de árbol y la utiliza para computar los gradientes mediante backpropagation. Este método nos permite agilizar el computo de los gradientes de las variables/parámetros deseados de forma que, partiendo del resultado de la función a diferenciar, aplicamos la regla de la cadena, y recorremos el grafo de computación de forma inversa para obtener los gradientes deseados en una única pasada [22].

A parte de los tensores -en los que hay que especificar de cuales hay que computar los gradientes-, es necesario elegir un algoritmo para la optimización y actualizar las variables de acuerdo con la dirección del gradiente natural. Como ya avanzábamos en la sección 2.2.3, vamos a utilizar el algoritmo Adam. Por fortuna, pytorch ya implementa este algoritmo como clase que permite la actualización de las variables mediante la llamada a la función *step()*. Por supuesto, esto deberá ir acompañado de la llamada a backward() sobre el tensor generado como resultado de nuestra función a optimizar [21].

Otras bibliotecas o soluciones como por ejemplo tensorflow, también hacen uso de tensores. Sin embargo, la forma de calcular los gradientes puede ser ligeramente diferente. Por ejemplo, en ten-

sorflow se realiza una aproximación del tipo 'plantilla', en el que el código escrito se va registrando generando un árbol de computación que contiene la metainformación. Posteriormente, a este árbol se le suministrarán datos para hacer los cálculos pertinentes. Por otro lado, pytorch utiliza un enfoque dinámico; es decir, cuando se realiza una operación esta se ejecuta y queda registrada en el árbol de computación, esto nos aporta una gran flexibilidad a la hora de desarrollar. Por supuesto, solo es necesario registrar en dicho árbol las variables y operaciones que afecten a nuestros gradientes.

4.2. Configuración inicial y ejecución.

El primer paso de la ejecución de nuestro programa es leer la configuración y aplicarla a nuestro programa para que se ejecute dentro del escenario deseado. Como se comentaba en el apartado 3.3.4, esto se hace en el módulo principal -*main*- de nuestro programa.

Después de realizar esta tarea, el programa entrará en un bucle para llamar a todas las pruebas especificados en la entrada del programa con sus correspondientes parámetros. Al finalizar cada prueba, imprimiremos, si procede, los tiempos devueltos. Cuando se terminen todos los tests se imprimirá un resumen de los warnings que han saltado durante la ejecución y se finalizará la rutina principal, esta tarea se puede observar en el código que se puede ver en el apéndice B.1.1.

4.3. Carga y particionado de datos.

Cuando vamos a realizar inferencia sobre un conjunto de datos predefinido, instanciar el dataset es el primer paso que debemos realizar. Esto se puede realizar de forma sencilla mediante la instrucción dataset = Dataset(params, generator=kernel). Por debajo, el código lo que hace es llamar al constructor, que leerá de un fichero en formato csv todos los atributos, la ruta al fichero se encuentra en el primer parámetro, que es un diccionario con los parámetros de entrada. La primera fila de este fichero se corresponde con los nombres de cada atributo y, aquellos atributos que delante del nombre tengan el símbolo '+', se utilizaran como atributos conocidos, y aquel que tenga el símbolo '-' será el que se trate de predecir. En el cuadro de código del apartado 4.3 se muestra el inicio de un dataset con este formato. Por último, solo nos queda mencionar que el módulo *Dataset* esta provisto de funcionalidad adicional que permite autogenerar un conjunto de datos unidimensional, para esto, en vez de llamar al dataset con la ruta al fichero hay que hacerlo una instancia de kernel, este sería el segundo parámetro de nuestra llamada.

Después de crear el dataset llega el momento de particionar los datos, como se adelantaba en la sección 3.3.3, tenemos dos estrategias disponibles: un particionado aleatorio de los datos mediante una instancia de la clase *RandomPartition*;o un particionado en el que se indique explícitamente los índices de los datos que se quieran usar para el entrenamiento, esto se consigue con una instancia

Código 4.1: Sintaxis de las cabeceras de un fichero de datos.

```
+Temperature,+ExhaustVacuum,+Pressure,+Humidity,-ElectricOutput␣
8.34,40.77,1010.84,90.01,480.48␣
23.64,58.49,1011.4,74.2,445.75␣
```

de *ExplicitPartition*. Por otro lado, cada una de estas clases implementa una función estática conocida como *batch_selector*, estas funciones son utilizadas por la clase correspondiente para seleccionar el conjunto de datos inicial $\overline{\mathbf{X}}$. De nuevo, la función *batch_selector* de *RandomPartition* escoge unos datos aleatorios; por el contrario, la de *ExplicitPartition* elige los datos de forma que sus índices sobre la partición de entrenamiento sean equidistantes. Es importante aclarar que, sea cual sea la estrategia de particionado, la función batch_selector que se va a utilizar internamente puede introducir por parámetros, de forma que sea pueda utilizar expresión lambda o cualquier otra estrategia ajena a la propia clase. La implementación de este módulo se puede encontrar en el apéndice B.1.2.

4.4. Funciones de kernel.

Una vez cargado y particionado nuestro dataset, podemos crear una instancia del kernel que vayamos a usar. En nuestro caso, Radial Basis Function o RBF, que mide la similitud entre datos utilizando la distancia entre los mismos como criterio. El enfoque que se ha tomado durante el desarrollo es el de tener una clase Kernel genérica que defina una interfaz común para diferentes funciones de kernel (aunque en este proyecto solo se use una). Esta interfaz define una función *process* que calcula la matriz de Gram del kernel definido en la subclase.

Esto ya lo habíamos realizado de forma similar a batch_selector en la partición (sección 4.3); es decir, la función se define de forma estática en la subclase y su constructor la carga en su superclase. Esta aproximación, quizá más orientado al paradigma funcional, nos permite sobrescribir la herencia e introducir pequeñas modificaciones sobre nuestra función. La decisión se ampara en la existencia de una literatura sobre las funciones de kernel que provee de definiciones con pequeñas variaciones (no esenciales) sobre la misma función. Recalcamos, como podemos ver en el código de la clase RBF en el apéndice B.1.3, que las variaciones están restringidas a los hiperparámetros definidos para ese kernel.

La interfaz de Kernel también definirá las operaciones suma y multiplicación para que sea posible combinar los kernels de acuerdo con la tesis de David K. Duvenaud [9]. El resultado de la operación será un nuevo kernel con las funciones de procesamiento de los operandos. Se pueden ver conjuntos de datos autogenerados mediante este proceso en la figura 5.1 de la sección 5.2.1. No profundizaremos más en esta dirección, nuestro único propósito era tener un esquema flexible para posibles ampliaciones.

En cuanto al kernel se refiere, solo nos queda comentar el propósito del argumento opcional de su constructor. Este parámetro es *fixed* y se encarga de habilitar el cálculo de los gradientes para los hiperparámetros, es decir, si en el caso de inferencia variacional quisiéramos encontrar nuestra distribución aproximada con unos hiperparámetros fijos, podremos hacerlo mediante esta opción. Otro caso de uso que nos permite esta variable es usar kernels compuestos donde uno de los kernels tiene hiperparámetros libres y el otro los tiene fijos.

La última pieza que nos queda por detallar de nuestro diseño es la clase *Hyperparameter*, como viene siendo habitual, el diseño está pensado para ser flexible y abstraer del resto del código las posibles restricciones que pueden tener nuestros hiperparámetros. En nuestro caso, se materializa en las clases *LengthScale*, *Noise* y *Variance*, donde la única restricción que tenemos es que sean tengan un valor mayor que cero. Esto representa un problema porque al usar el optimizador de pytorch nuestros tensores pueden entrar en el dominio de los números negativos con facilidad. La solución propuesta, tal y como se implementa en el otro código del apartado 4.4 y que se puede ver de forma completa en apéndice B.1.3, consiste en transformar nuestros valores iniciales a un espacio logarítmico, estos serán los valores que optimicemos, y cuando necesitemos nuestros parámetros para operar en el kernel, recuperaremos su valor computando la exponencial de nuestro tensor.

Código 4.2: Fragmento de código de la clase *RBFCommon* que transforma al espacio logaritmico los hiperparámetros del kernel gaussiano.

```
77      def transform(self,␣value):␣
78          return t.log(t.tensor(value,dtype=self.dtype)).item()␣
79
80      def getValue(self):␣
81          v␣=␣t.exp(self.value)␣
82          Logger.logDebug(str(self)+"␣=>␣"+␣str(v.item()),[],␣"Hyperparam.getValue")␣
83          return v␣
```

4.5. Inferencia usando procesos gaussianos.

Ahora que ya podemos crear particiones e instancias de kernel, tenemos todos los ingredientes necesarios para realizar inferencia sobre el conjunto de datos. Como venimos explicando desde el capítulo 2.2.3, esto lo podemos realizar de dos formas, utilizando un proceso gaussiano o su versión aproximada. A continuación, exponemos las dos.

Procesos Gaussianos

Los procesos gaussianos los vamos a definir en la clase *GaussianProcess*, y aunque en su código podemos ver métodos para inicializar, normalizar, calcular medias, etc. Lo cierto es que las rutinas

principales son:*hyper_optim()*, *train()* y *test()*.

En primer lugar, el método *hyper_optim()*, lo vamos a utilizar para obtener los hiperparámetros que maximicen la verosimilitud marginal de acuerdo con lo expuesto en la sección 2.1.2. En segundo lugar, en el método train(), vamos a realizar todos los cálculos que dependen de nuestras observaciones para obtener los resultados dados por las ecuaciones 2.6. Es importante que ahora recordemos aquellas propiedades de la distribución gaussiana -marginalización, producto, etc.-; ya que son las que nos van a permitir realizar determinados tipos de operaciones matriciales (multiplicación element-wise, etc.) para calcular nuestro vector de medias y matriz de covarianzas. Finalmente, en el método *test()* obtendremos la solución aplicando las mismas ecuaciones en los puntos en los que queramos realizar inferencia. Los códigos referentes a estos métodos pueden verse en el apéndice B.1.4.

Procesos Gaussianos Aproximados

En cuanto a los procesos gaussianos aproximados, el proceso de entrenamiento tiene algunos cambios. En concreto, las técnicas expuestas en la sección 2.2.3, nos van a permitir hacer el entrenamiento y la optimización de forma conjunta y segmentar nuestra entrada de datos reduciendo sustancialmente la cantidad de memoria que tenemos reservada en un instante dado. En cuanto al test, este se hace de forma muy similar al del proceso gaussiano, pero en este caso tendremos el vector de medias y matriz de covarianzas definidos en la ecuación 2.11, y la optimización la llevaremos a cabo mediante la aproximación por minibatches de la ecuación 2.17. El código relativo a estos métodos puede verse en el apéndice B.1.4.

4.6. Calidad de ajuste.

La última pieza que necesitamos añadir a nuestro entorno es el módulo de métricas *Metric*. Su propósito es obtener una medida de la calidad del ajuste para ello, y como veíamos en la sección 3.3.4, existen diferentes criterios para obtener nuestras métricas. El funcionamiento general es muy sencillo, se carga un objeto de tipo *Metric* en nuestro proceso gaussiano. Como se puede ver en el apéndice B.1.5, podemos instanciar varias métricas dentro de un objeto Metrics especificando, cada cuantas iteraciones de actualización de los hiperparámetros se va a calcular cada una de ellas.

Como detalles adicionales podemos mencionar que los objetos Metric tienen acceso al objeto del que están calculando las métricas, esto es para los casos en los que una métrica requiera invocar alguna funcionalidad adicional de dicho objeto. Por ejemplo, el error cuadrático medio, necesita los resultados de la rutina *test()*.

Por último y por defecto, los valores de cada métrica se mostrarán en relación con el momento (tiempo) en el que se han generado, esto se realiza gracias a un temporizador que se inicializa en la primera ejecución.

5 Integración, Pruebas y Resultados

5.1. Requisitos del sistema.

Requisitos Software.

La ejecución de todo el software desarrollado para este trabajo se ha realizado sobre un sistema GNU/Linux Debian 10.5. Las dependencias de software se exponen en las tablas 5.1 y 5.2. Es posible, que alguna de estas dependencias, como la del sistema operativo, se puedan satisfacer de otras maneras. Sin embargo, el software no se ha probado fuera de este entorno. Garantizar el correcto funcionamiento de nuestro software en estos escenarios queda, por lo tanto, fuera de nuestro alcance.

El software y drivers utilizados

Software	Versión	Instalación
Python	3.7	Gestor de paquetes.
Pip	3	Gestor de paquetes.

Tabla 5.1: En esta tabla se detalla el software utilizado.

Por su utilidad, y de forma opcional, puede resultar interesante el uso de los siguientes paquetes:

- *fim*: para automatizar el visionado de las posibles imágenes generadas.
- *nvtop*: para la monitorización de los recursos que se utilizan en la tarjeta gráfica en tiempo de ejecución.
- *htop*: para la monitorización de los recursos que utilizan de memoria y cpu en tiempo de ejecución.
- *anaconda*: Alternativa a pip que permite crear entornos virtuales en los que se puedan probar diferentes configuraciones de los paquetes.

Requisitos Hardware.

Las prestaciones hardware sobre las que se han ejecutado las pruebas detalladas en el apéndice B.1.6, consisten en: microprocesador a 3.20GHz, 16GB RAM, una tarjeta gráfica de Nvidia con 8 GB de

Módulos Python	Versión
Pytorch	1.6.0+cu92
Matplotlib	3.3.3
pandas	1.1.5
numpy	1.19.4

Tabla 5.2: En esta tabla se detallan los módulos de python utilizados. La instalación se lleva a cabo mediante pip, y el paquete de pytorch, en concreto, utilizando las opciones especificadas en la documentación oficial [21].

RAM y 2560 núcleos CUDA. Técnicamente ningún requisito es imprescindible para obtener resultados; sin embargo, para acceder a ciertas funcionalidades como el uso de tarjeta gráfica, o para replicar las pruebas con la misma configuración de parámetros del programa (cantidad de datos por batch, etc.), si pueden serlo. Por esta razón solo especificamos las líneas generales que siguen el hardware utilizado.

Estructura de directorios del paquete.

Los ficheros de código que implementan este proyecto se organizan en una estructura de paquete de tipo python (cada directorio de código con su __init__.py). En el directorio raíz tenemos el lanzador del programa y luego los siguientes directorios: inference, covariances, data, utilities, tests, input y output. Los nombres son auto explicativos, y sirven para guardar los ficheros con la funcionalidad especificada en la sección 3.2. Los directorios restantes son para almacenar los ficheros de datos y las rutas por defecto para guardar la salida del programa. En el apéndice C se puede ver el directorio utilizado para el proyecto.

5.2. Pruebas realizadas y resultados obtenidos.

La funcionalidad del código desarrollado la vamos a explotar mediante un conjunto de pruebas. Estas servirán para comprobar el correcto funcionamiento del código y medir los rendimientos de las rutinas principales. Todas las pruebas desarrolladas para este trabajo se pueden ver en apéndice B.1.6, aquí nos centraremos en desarrollar aquellos que abordan las características principales de nuestro programa.

5.2.1. Generación de datos y funciones de kernel.

El primer test que vamos a realizar es para generar un conjunto de datos mediante un kernel gaussiano. Podemos ver el fragmento de código utilizado en test1 (línea 23) del apéndice B.1.6.

Como podemos ver, lo que estamos haciendo es instanciar un kernel gaussiano que será utilizado para generar los datos. En nuestro código, generamos un vector aleatorio de muestras de una distribución gaussiana $N(\mu = 0, \sigma^2 = 1)$ y las multiplicamos por la descomposición de Cholesky del kernel generado a partir de nuestra entrada. Es un caso simple que nos permite obtener un proceso gaussiano univariante.

Si ejecutamos este código utilizando el fichero de configuración por defecto (apéndice B.3) para generar conjuntos de 90 datos, obtenemos la salida mostrada en la figura 5.1.

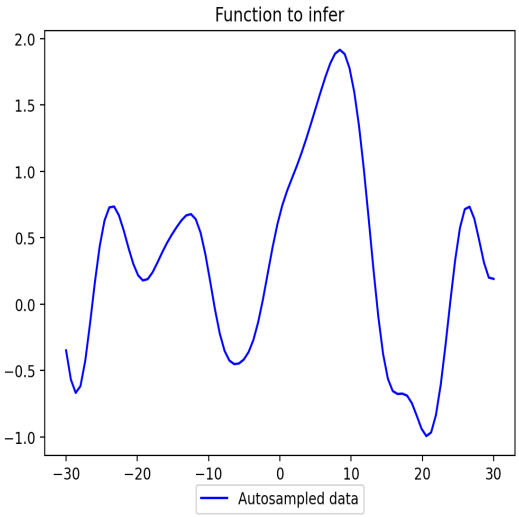


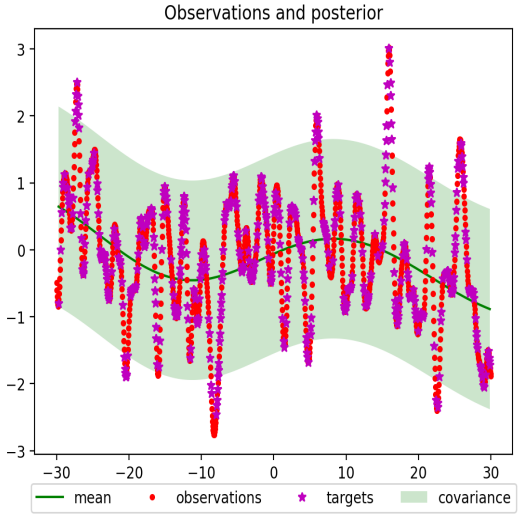
Figura 5.1: En la imagen, podemos ver el conjunto de datos generado con la función de kernel RBF. Estas imágenes han sido generadas mediante el test 1 del código del apéndice B.1.6.

5.2.2. Carga de datos, particionado y ajuste del proceso gaussiano a los datos.

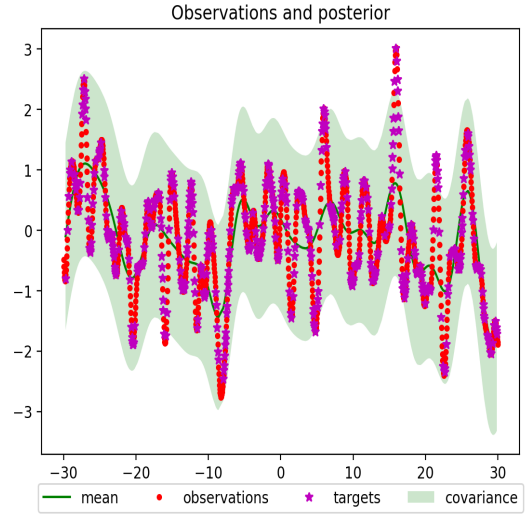
En este test, cuyo código se muestra en el test 6 del apéndice B.1.6 (línea de código 131), engloba algo más de funcionalidad. En primer lugar, ahora en el input no restringimos explícitamente la carga de datos de fichero. En segundo lugar, ahora particionamos nuestros datos en conjunto de entrenamiento y de test. Finalmente crearemos un proceso gaussiano y elegiremos los parámetros óptimos para que se ajuste de la mejor forma posible a los datos; esto se hace de acuerdo con el criterio de máxima verosimilitud presentado en la sección 2.1.2.

Este test, lo vamos a ejecutar utilizando el fichero de configuración por defecto presente en el

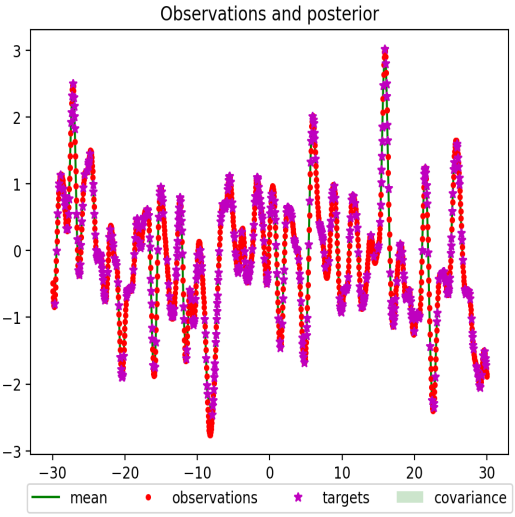
apéndice B.3. En la figura 5.2, comparamos las soluciones obtenidas de entrenar durante un diferente número de épocas.



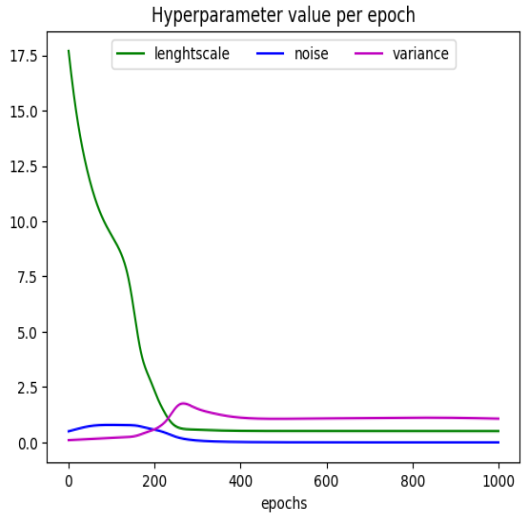
(a) Optimización usando 10 épocas.



(b) Optimización usando 300 épocas.



(c) Optimización usando 1000 épocas.



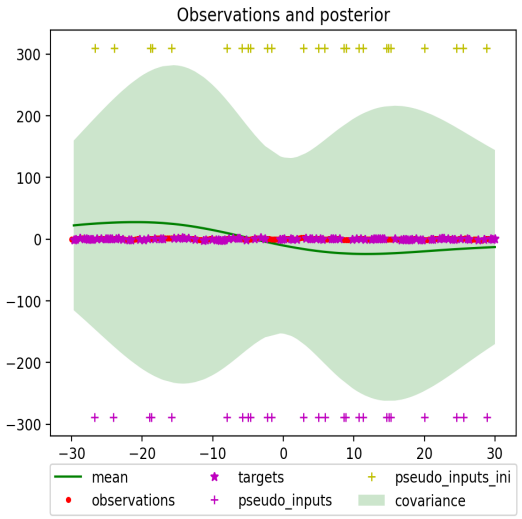
(d) Cambios en los hiperparámetros durante 1000 épocas.

Figura 5.2: En las subfiguras 5.2(a), 5.2(b) y 5.2(c) aparece un proceso gaussiano entrenado durante 10, 300 y 1000 épocas respectivamente. En la subfigura 5.2(d), vemos como varían los hiperparámetros durante cada época para un kernel generador con $(l, n, v) = (0{,}5, 1e-8, 1)$ y una inicializacion $(l, n, v) = (17{,}43, 0{,}5, 0{,}1)$. El código que las ha generado está en el apéndice B.1.6.

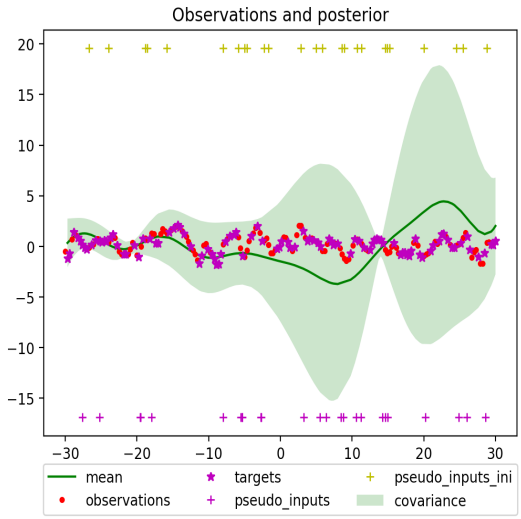
5.2.3. Aproximando nuestro proceso gaussiano.

En el test que vamos a mostrar ahora introducimos nuestro proceso gaussiano aproximado. Dado que la interfaz que tenemos es la misma que para un proceso gaussiano normal, el código, presente en el test 7 del apéndice B.1.6 (línea de código 174), es muy similar al presentado en la sección 5.2.2.

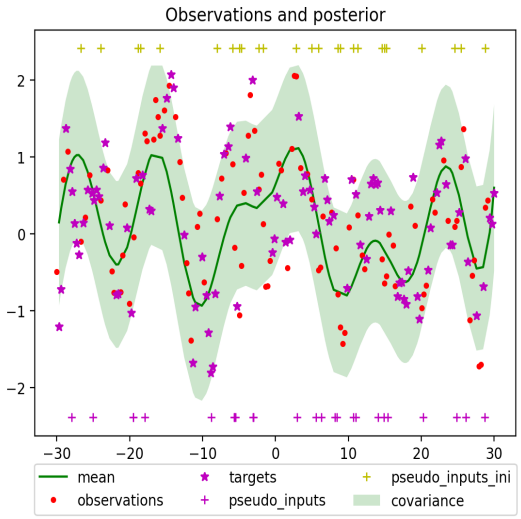
En nuestra ejecución utilizamos la configuración del fichero por defecto (apéndice B.3). La salida obtenida la mostramos en la figura 5.3. Como se puede ver, para simplificar la visualización, volvemos a utilizar un conjunto autogenerado de datos.



(a) Optimización usando 10 épocas.



(b) Optimización usando 500 épocas.



(c) Optimización usando 1000 épocas.

Figura 5.3: En las subfiguras 5.3(a), 5.3(b) aparece un proceso gaussiano aproximado entrenado durante 10, 500 y 1000 épocas respectivamente. El código que las ha generado está en el apéndice B.1.6.

5.2.4. Métricas de calidad de ajuste.

Como novedad, y como ahora nos vamos a centrar en las métricas de calidad del ajuste, vamos a cargar nuestros primeros conjuntos de datos. Dichos conjuntos proceden del repositorio de UCI, y

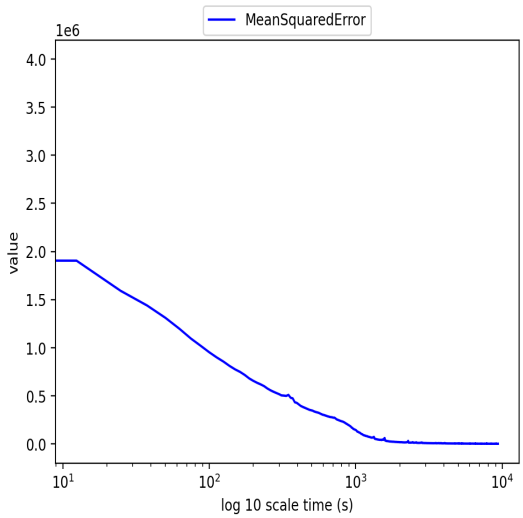
los vamos a utilizar en consonancia con los procesos gaussianos. Es importante hacer notar que es complicado replicar algunas de estas pruebas utilizando procesos gaussianos o con prestaciones de hardware inferiores a las especificadas en la sección 5.1. Esto es debido principalmente a la gran cantidad de datos con la que vamos a trabajar. Tal y como podemos ver en el test 8 del código del apéndice B.1.6 (línea 208), vamos a obtener el Error Cuadrático Medio, El logaritmo de la verosimilitud con los datos de test y la cota mínima calculada según la sección 2.2.3.

La invocación de esta prueba se va a realizar de acuerdo con las configuraciones usadas en el script autorun.sh (apéndice B.2) para el test 8. El dataset utilizado ha sido el 'YearPredictionMSD' [4], este consta con 505345 instancias de 91 atributos, 90 de los cuales sobre mediciones del timbre de la voz en canciones y el último -y que se pretende inferir- del año en el que se publicó la canción. La salida de la ejecución, la podemos en la figura 5.4.

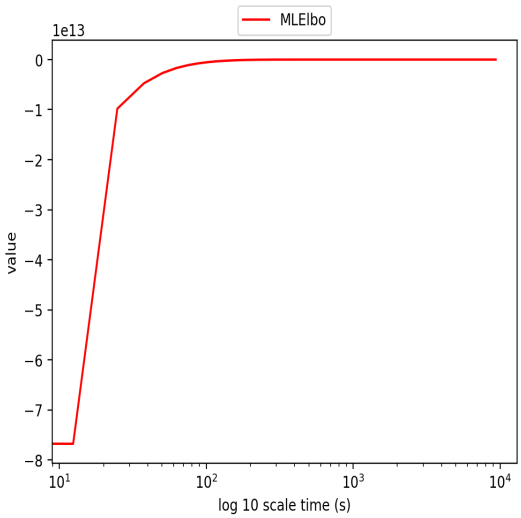
5.2.5. Midiendo el rendimiento.

Finalmente, y quizá el principal propósito de nuestro trabajo consiste en medir el rendimiento de nuestro algoritmo. Para ello vamos a diseñar unas pruebas que, utilicen el código ya existente, para calcular el coste temporal de nuestros procesos gaussianos y procesos gaussianos aproximados. Esto lo haremos midiendo el tiempo a la vez que variamos el tamaño de nuestra entrada de datos, o habilitamos el uso de la GPU. Por su extensión, el código solo lo incluiremos en el apéndice B.1.6 (línea 286).

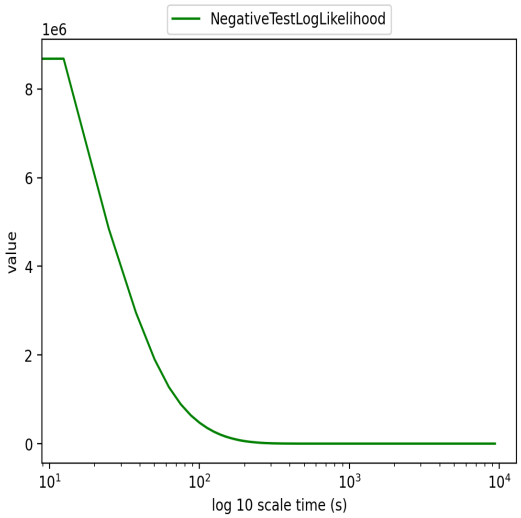
En la figura 5.5 podemos ver los resultados de ejecutar las pruebas relativas a los tests 10 y 11 del script autorun.sh del apéndice B.2. Debido a las limitaciones de memoria del proceso gaussiano, no podemos incrementar demasiado el tamaño de la entrada, por esta razón utilizaremos el dataset 'Physicochemical Properties of Protein Tertiary Structure Data Set' [4], que tiene 45730 instancias de datos con 10 atributos -uno para inferir-, sobre las propiedades de este tipo de proteínas, cada una. No obstante, cómo podemos ver en la figura 5.6, podremos volver a recuperar el dataset 'YearPredictionMSD' [4] en el proceso gaussiano aproximado. Como observación, podemos destacar que la frecuencia de procesamiento de la CPU es más alta que la de la GPU, por esta razón, cuando no hay muchos datos puede procesar más rápido esta primera. Además, también resulta curioso ver como a medida que crece el número de datos en el caso aproximado, el tiempo se reduce. Esto es debido a que cuando aumentamos el tamaño del batch estamos, como consecuencia, reduciendo el número de épocas. No obstante, el tiempo de ejecución por batch si se incrementa, aunque no lo suficiente como para tener un impacto negativo. Esto lo podemos ver de nuevo en la figura 5.6.



(a) YearPredictionMSD MSE para proceso gaussiano aproximado.

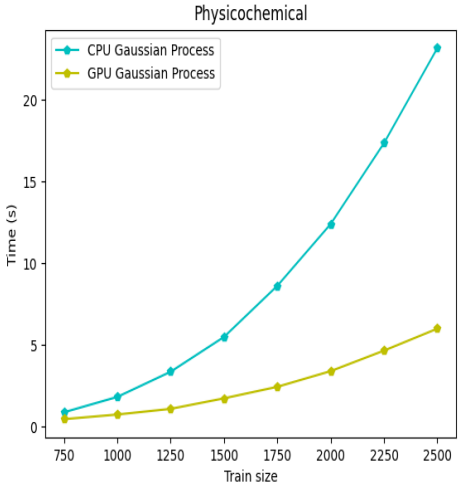


(b) YearPredictionMSD ELBO para proceso gaussiano aproximado.

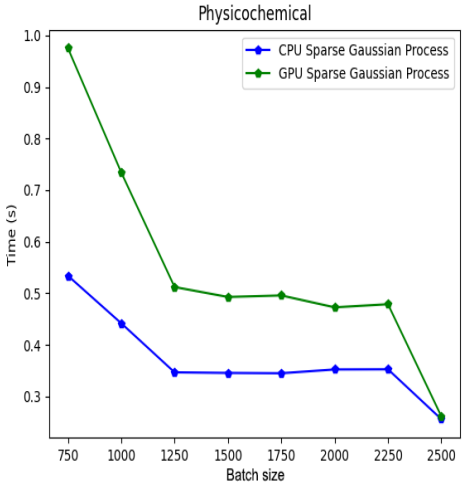


(c) YearPredictionMSD test loglikelihood para proceso gaussiano aproximado.

Figura 5.4: En las subfiguras 5.4(a), 5.4(b) y 5.4(c) se corresponden con el el MSE, ELBO y test loglikelihood para el dataset YearPredictionMSD. El código que las ha generado esta en el apéndice B.1.6, y la invocación al programa ha sido realizada con las siguientes opciones de entrada: '−−cfile "input/conf/year_prediciton_conf.json"', '−−epochs 750', '−−gpu', '−t "8"' y '−−batch−trsize 1000'.

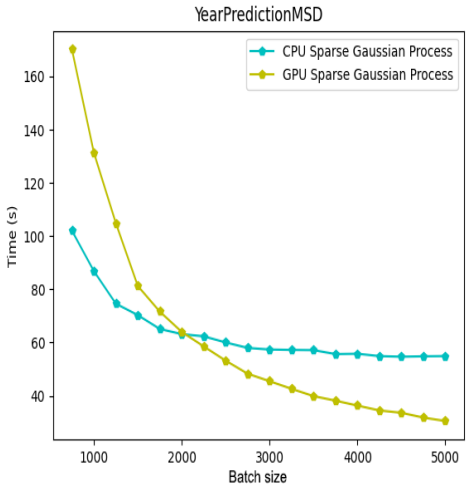


(a) Rendimiento GP con Physicochemical.

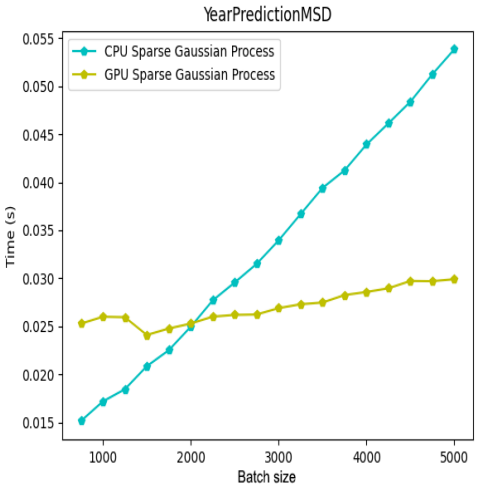


(b) Rendimiento SGP con Physicochemical.

Figura 5.5: En la subfigura 5.5(a), podemos ver el tiempo que tardamos en ajustar el proceso gaussiano (GP) a los datos (10 épocas) a medida que vamos incrementando el tamaño de la entrada, esto se hace para GPU y CPU. En la subfigura 5.5(b), tenemos la misma representación para un proceso gaussiano aproximado (SGP), en esta ocasión aumentando el tamaño del batch y reduciendo el número de estos en consonancia; dado que los costes escalan a menor ritmo y se habilita la paralelización, el tiempo de procesamiento por batch aumentará (figura 5.6), pero el total se reducirá. El dataset utilizado para ambas es 'Physicochemical Properties of Protein Tertiary Structure Data Set' [4] y el comando de generación se puede ver en el apéndice B.2.



(a) Rendimiento con YearPredictionMSD.



(b) Rendimiento tamaño/(batch*epoca) con YearPredictionMSD.

Figura 5.6: En la subfigura 5.6(a) se representa el coste computacional medio para procesar un batch (variando su tamaño) del proceso gaussiano aproximado (SGP). La 5.6(b), es la misma representación, pero también se relativiza al nº de batches, que decrece, si sube su tamaño. El comando de generación, que utiliza el dataset YearPredictionMSD [4], se puede ver en el apéndice B.2.

6 Conclusiones y Trabajo Futuro

6.1. Conclusiones.

El objetivo de este trabajo siempre ha sido comprobar en el plano práctico nuestros cálculos teóricos, así como estudiar el impacto que tienen diferentes tecnologías en el cómputo de nuestro problema. En particular, hemos aprendido a utilizar la librería Pytorch para diseñar y desarrollar un *framework* que nos permita: realizar inferencia implementando como modelo los procesos gaussianos y su versión aproximada, así como lograr abstraer la ejecución de los algoritmos de las diferentes unidades de procesamiento (CPU y GPU). Por supuesto, aquí se incluye el desarrollo de un sistema de métricas para medir los costes y la calidad del ajuste, esto nos ha permitido obtener resultados de entornos reales como el de los datasets *YearPredictionMSD* y *Physicochemical Properties of Protein Tertiary Structure*.

En términos computacionales, y como veíamos en la sección 5.2.5, tanto la aproximación del posterior, como la utilización de la GPU, tienen un impacto positivo en la resolución de nuestro problema. En consecuencia, obtenemos la capacidad para computar conjuntos de datos que no podíamos con las soluciones tradicionales: posterior exacto y CPU. Esto, que se ilustra muy bien en las figuras 5.5 y 5.6, nos permiten ver como el modelo aproximado es capaz de tratar mayor cantidad de datos en menos tiempo, y como la GPU acelera el procesamiento; sobre todo, cuando aumentamos el tamaño del minibatch. Por un lado, esto es debido a la disminución del tamaño de $\mathbf{K}$; y por el otro, al aumentar la paralelización, mantenemos más estable el tiempo de ejecución por batch a la vez que reducimos el número de épocas. Dado que el coste computacional de computar un batch no aumenta más de lo que nos ahorramos al reducir el número de épocas, conseguimos disminuir el tiempo total de computación.

6.2. Trabajo Futuro.

En cuanto al trabajo futuro, existen diferentes caminos que podemos tomar, aquí se van a proponer dos. El primero es ampliar la riqueza de métodos implementados: ríos de tinta se han vertido sobre los procesos gaussianos, sus diferentes aproximaciones y las funciones de kernel. Por poner un ejemplo,

los diferentes métodos expuestos en [7]. Otra opción es añadir soporte para clasificación, en el caso binario -regresión logística-, necesitaremos cambiar la verosimilitud ($\mathcal{N}(\mathbf{y}|\mathbf{f}(\mathbf{x}), \sigma_n^2)$) por una función, denominada sigmoide ($sigmoide(f(x)y)$), cuyo output esté restringido al dominio binario $[0, 1]$, y cuya problemática se extenderá hasta la aproximación de la esperanza, donde habrá que usar cuadratura en una dimensión [8]. Esto se puede desarrollar incluso más resolviendo problemas multiclase, aunque en este caso la complejidad será superior debido a que necesitaremos un GP por cada atributo [8].

El segundo camino se va a centrar en la computación distribuida. Pytorch es una librería que nos permite paralelizar la computación en distintas máquinas y dispositivos (CUDA), en este trabajo esta es una de las cuestiones que no se ha tocado. Sin embargo, en términos de reducción de coste temporal puede ser una solución interesante, sobre todo ahora que la tecnología cloud está viviendo un auténtico *boom*.

Bibliografía

[1] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," *NIPS'05: Proceedings of the 18th International Conference on Neural Information Processing*, 2005.

[2] M. K. Titsias, "Variational learning of inducing variables in sparse gaussian processes," *Volume 5 of JMLR:W&CP 5*, 2009.

[3] J. Hensman, N. Fusi, and N. D. Lawrence, "Gaussian processes for big data," *arXiv:1309.6835*, 2013.

[4] "Uci dataset repository." https://archive.ics.uci.edu.

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*. Information Science and Statistics, Springer, 2011.

[6] K. P. Murphy, *Machine Learning A Probabilistic Perspective*. Adaptive Computation and Machine Learning, The Mit Press, 1 ed., 2012.

[7] J. Quiñonero-Candela and C. E. Rasmussen, "A unifying view of sparseapproximate gaussian process regression," *Journal of MAchine Learning Research 6*, vol. Journal of Machine Learning Research 6 (2005), 2005. Editor:Ralf Herbrich.

[8] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning, The MIT Press, 2006.

[9] D. K. Duvenaud, *Automatic Model Construction with Gaussian Processes*. PhD thesis, University of Cambridge, 2014.

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning:An Introduction*. The MIT Press, 2005.

[11] "https://www.internetlivestats.com/." web, 2 2021.

[12] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, ch. 1, p. 2. Adaptive Computation and Machine Learning, The MIT Press, 2006.

[13] J. de la Horra Navarro, *Estadística aplicada (Spanish Edition)*. Díaz de Santos, 2012.

[14] E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[15] S. Roweis, "Gaussian identities," July 1999.

[16] K. P. Murphy, *Machine Learning A Probabilistic Perspective*, ch. 14. Adaptive Computation and Machine Learning, The Mit Press, 1 ed., 2012.

[17] K. P. Murphy, *Machine Learning A Probabilistic Perspective*, ch. 15. Adaptive Computation and Machine Learning, The Mit Press, 1 ed., 2012.

[18] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *Journal of MAchine Learning Research*, 2013.

[19] A. G. de G. Matthews, J. Hensman, R. E. Turner, and Z. Ghahramani, "On sparse variational methods and the kullback-leibler divergence between stochastic processes," 2015.

[20] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *ICLR 2015*, 2015.

[21] *Pytorch Documentation - https://pytorch.org/docs/1.6.0/*. https://pytorch.org/docs/1.6.0/.

[22] C. Olah, “Calculus on computational graphs: Backpropagation.” web, Aug. 2015. https://colah.github.io/posts/2015-08-Backprop/.

[23] D. J. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2005.

Apéndices

A

Conceptos teóricos

A.1. Regla de Bayes

Teorema o Regla de Bayes [23]:

Sean x_1, ..., x_n sucesos tales que:

1.– Su unión conforme el espacio muestral: $\cup_{i=1}^{n} A_i = \Omega$

2.– Sean independientes entre si: $A_i \cap A_j = \emptyset$ para todo $i \neq j$

Entonces, usando las reglas de probabilidad condicional, suma y del producto en [6] [5] [23], el Teorema o Regla de Bayes surge de forma natural:

$$p(y|x, H) = \frac{p(x|y, H)p(y|H)}{\sum_{y'} p(x|y', H)p(y'|H)} \tag{A.1}$$

B

Código

B.1. Código para realizar inferencia.

B.1.1. Lanzador del programa.

```
1  import datetime
2  import torch as t
3  import sys
4  import time
5  from tests.test_set import *
6  from tests.optim_tests import *
7  from utilities.logger import Logger
8  from utilities.input import conf_load
9
10 if __name__ == "__main__":
11
12     # load parameters
13     params = conf_load(sys.argv[1:])
14     Logger.logInfo("Loading parameters")
15
16     #Program configuration options
17     conf = {"log_level": params["log_level"],
18             "log_mode":  params["log_mode"],
19             "log_dmode": params["log_dmode"],
20             "log_dmask": params["log_dmask"],
21             "colorless": params["colorless"]
22             }
23     Logger.setLogConf(conf)
24     try:
25         t.manual_seed(params["seed"])
26         Logger.logInfo("Setting seed defined by user")
27     except:
28         params["seed"] = t.seed()
29         t.manual_seed(params["seed"])
30         Logger.logWarning("Autogenerating seed")
31
```

```
32
33      d = datetime.datetime.now()
34      date = str(d.year) + "-" + str(d.month) + "-" + str(d.day) + "-" \
35             + str(d.hour) + "-" + str(d.minute) + "-" + str(d.second)
36
37      #pytorch configuration options
38      Logger.logInfo("Setting pytorch configuration options")
39      t.set_printoptions(threshold=100000)
40      t.autograd.set_detect_anomaly(True)
41      if(params["gpu"]):
42          if (t.cuda.is_available()):
43              dev = t.cuda.current_device()
44              count = t.cuda.device_count()
45
46              capa = t.cuda.get_device_capability() #Default gets current_device
47              name = t.cuda.get_device_name()
48              if(count > 1):
49                  Logger.logWarning("Found " + count + "aditional device/s")
50              t.cuda.device(dev)
51              Logger.logInfo("Using " + name)
52              params["device"] = t.device("cuda")
53          else:
54              Logger.logWarning("Cuda is not available. Using CPU.")
55              params["device"] = t.device("cpu")
56
57      else:
58          Logger.logInfo("Using CPU.")
59          params["device"] = t.device("cpu")
60
61      if("nthreads" in params):
62          nthreads = t.get_num_threads()
63          if(params["nthreads"] > nthreads or params["nthreads"] < 1):
64              Logger.logWarning("There are " +str(nthreads)+" available threads,"+\
65                                "but you tried to use "+ str(params["nthreads"]))
66              Logger.logWarning("Using all threads by default")
67          else:
68              Logger.logInfo("Using "+str(params["nthreads"])+" nthreads.")
69              t.set_num_threads(params["nthreads"])
70              t.set_num_interop_threads(params["nthreads"])
71
72      #Testing
73      tests = {
74              #Functionality tests
75              "1":("Autosampling Gaussian Kernel",       test_autosampling_rbf),
76              "2":("Autosampling Periodic Kernel",       test_autosampling_sin),
77              "3":("Autosampling Gaussian + Periodic",   test_autosampling_rbf_sin),
78              "4":("Autosampling(RandomPartition)",      test_random_partition),
79              "5":("Autosampling(ExplicitPartition)",    test_explicit_partition),
```

```
            "6":("GP Hyperparameter Optimization",      test_hyperp_optimization),
            "7":("Sparse Gaussian Process",             test_sgp),
            "8":("Sparse Gaussian Process Metrics",     test_sgp_metrics),
            #Performace tests
            "9":("GaussianProcess performance",         test_gp_performance),
            "10":("GaussianProcess optimization"+\
                  " performance",                       test_gphyperp_performance),
            "11":("Sparse Gaussian Process"+\
                 "performance",                         test_sgp_performance),
            "12":("Optim changes",                      test_optim)}


    for n in params["tests"].split(","):
        t.manual_seed(params["seed"])
        try:
            Logger.logInfo("Starting test " + tests[n][0])
            params["test_name"] = tests[n][0] + "_" + date
        except:
            Logger.logError("Bad number of test input: " + n)

        Logger.setLogDir(params["log_dir"]+params["test_name"])
        #Executing test
        chrono = time.time()
        res = tests[n][1](params)
        total_time = time.time() - chrono

        res_msg = "\n"+\
                  "}_____________ Execution Summary _____________{\n"+\
                  "   => Total time: " + str(total_time)
        for k,v in res.items():
            res_msg += "\n     - " + k + ": " + str(v)
        res_msg+="\n\n"
        Logger.logInfo(res_msg)

    Logger.warningSummary()
```

B.1.2. Carga y particionado de datos.

Dataset

```
import pandas as p
from utilities.logger import Logger
import torch as t
from utilities.matrix_ops import checkMatrixProperties
from data.partition import ExplicitPartition as explicit
import matplotlib.pyplot as plot

class Dataset:
```

```
 9     def __init__(self, params, generator=None, reduc = 0):
10         self.dtype = params["dtype"]
11         self.device = params["device"]
12         self.random_variables=None
13         self.generator = generator
14         self.validation = True
15         data = None
16 
17         size = params["xstarn_data"]
18         s_min, s_max = params["scope_min"], params["scope_max"]
19         if("dataset_file" in params):
20             try:
21                 Logger.logInfo("Loading [*] dataset",values=[params["dataset_file"]])
22                 csv_file = p.read_csv(params["dataset_file"])
23                 data = self.parser(csv_file)
24             except Exception as e:
25                 if(generator == None):
26                     Logger.logError("Dataset[" + params["dataset_file"] + "] doesn't
                           exist.",
27                                     "Dataset.__init__")
28                 else:
29                     Logger.logWarning("Dataset[" + params["dataset_file"] + "] doesn'
                           t exist.",
30                                       "Dataset.__init__")
31                     Logger.logWarning("Autosampling a dataset.", "Dataset.__init__")
32                     data = self.auto_sampling(size, s_min,s_max)
33 
34         elif(generator != None):
35             Logger.logInfo("Autosampling a dataset.")
36             data = self.auto_sampling(size, s_min,s_max)
37         else:
38             Logger.logError("Bad dataset input and no generator found.",
39                             "Dataset.__init__")
40 
41         self.random_variables = data
42         if(reduc > 0): self.reduce(reduc)
43 
44         Logger.logInfo("A dataset with {Instances = *, train atributes = *,"+\
45                        "test atributes = *} is ready.","",[self.size(), self.
                           getXDimensionality(),
46                        self.getYDimensionality()])
47 
48     def reduce(self, n):
49         Logger.logInfo("Reducing input dataset...")
50         if(self.generator != None):
51             Logger.logWarning("Autosampled datasets can't be reduced.")
52             return
53         indices = explicit.batch_selector(self.size(), n)
```

```
54
55          x = self.random_variables["xdata"]
56          tem_x = t.empty(n, self.getXDimensionality(), dtype=x.dtype)
57          tem_x = self.random_variables["xdata"][indices]
58          del self.random_variables["xdata"]
59          self.random_variables["xdata"] = tem_x
60
61          tem_y = t.empty(n, self.getYDimensionality(), dtype=x.dtype)
62          tem_y = self.random_variables["ydata"][indices]
63          del self.random_variables["ydata"]
64          self.random_variables["ydata"] = tem_y
65
66
67      def size(self, i=0):
68          return self.random_variables["xdata"].size()[0]
69
70      def getData(self,i=0):
71          return self.random_variables["xdata"], self.random_variables["ydata"]
72
73      def getXDimensionality(self):
74          return self.random_variables["xdata"].size()[1]
75
76      def getYDimensionality(self):
77          return self.random_variables["ydata"].size()[1]
78
79      def parser(self, r):
80          y_atribs = []
81          x_atribs = []
82          count_x=0
83          count_y=0
84          dicc = {}
85          x_acc = None
86          y_acc=None
87
88          #Iterate over different columns
89          for col in r.columns:
90              #Create a new tensor of dimension [length(col), 1]
91              new_col = t.tensor(r[col].values, dtype=self.dtype, device=self.device)
92              new_col = new_col.reshape(len(r[col].values),1)
93              #If attibute name contains +: x data
94              if "+" in col:
95                  x_atribs.append(col)
96                  if(count_x == 0):
97                      x_acc = new_col
98                  else:
99                      x_acc = t.cat((x_acc, new_col),1)
100                 if(True in t.isnan(x_acc)):
101                     Logger.logError("Dataset contains train NANs value")
```

```
102                 count_x += 1
103             #If attibute name contains -: y data
104             elif "-" in col:
105                 #col name
106                 y_atribs.append(col)
107                 if(count_y == 0):
108                     y_acc = new_col
109                 else:
110                     y_acc = t.cat((y_acc, new_col),1)
111                 count_y += 1
112                 if(True in t.isnan(y_acc)):
113                     Logger.logError("Dataset contains test NANs value")
114 
115         dicc["xatrib"] = x_atribs
116         dicc["yatrib"] = y_atribs
117         dicc["xdata"] = x_acc
118         dicc["ydata"] = y_acc
119 
120         count=y_acc.size()[0]
121 
122         self.x_dimensionality = count_x
123         self.y_dimensionality = count_y
124 
125         return dicc
126 
127     def getAsymetric(self):
128         return self.asymetric_data
129 
130     def auto_sampling(self, n_data, scope_min=-5, scope_max=5):
131         dicc = {}
132         x = t.linspace(scope_min, scope_max,n_data, dtype = self.dtype, device = self
                .device)
133         x = x.reshape(x.size()[0],1)
134         Kxx = self.generator.process(x)
135         L = t.cholesky(Kxx)
136         y = t.matmul(L, t.normal(0,1, (1,n_data),dtype=self.dtype,device=self.device)
                [0])
137         dicc["xatrib"] = "Autosampled"
138         dicc["yatrib"] = "Autosampled"
139         dicc["xdata"] = x
140         dicc["ydata"] = y.reshape(y.size()[0],1)
141 
142         return dicc
143 
144     def plot_sample(self, path,dpi):
145         x, y = self.getData()
146         if(x.size()[1] > 1):
147             Logger.logWarning("Objetive function plot only supports 1-feature data")
```

```
148             return
149         f = plot.figure()
150         plot.plot(x.cpu(), y.cpu().detach().numpy(),'b-' ,label='Autosampled data')
151         plot.title("Function to infer")
152         legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,-0.05), ncol=1)
153         f.savefig(path, dpi=dpi, bbox_extra_artists = [legend],bbox_inches="tight")
```

Partition

```
1  import torch as t
2  from utilities.logger import Logger
3  class Partition:
4      def __init__(self, dataset, selector):
5          self.dataset = dataset
6          self.train_indexes, self.test_indexes = self.select()
7          self.selector = selector
8
9          if (dataset.size() < self.training_size):
10             Logger.logError("Dataset size ("+str(dataset.size())+\
11                             ") < train size("+str(self.training_size)+")")
12
13     def select(self):
14         pass
15
16     def get_sparse_indexes(self):
17         return self.sparse_indexes
18
19     def setBatchSelector(selector):
20         self.selector = selector
21
22     def getTrainData(self):
23         xdata, ydata = self.dataset.getData()
24
25         train_xdata = xdata[self.train_indexes]
26         train_ydata = ydata[self.train_indexes]
27
28
29         return {"Xtrain":train_xdata,"Ytrain":train_ydata}
30
31     def getTestData(self):
32         xdata, ydata = self.dataset.getData()
33
34         test_xdata = xdata[self.test_indexes]
35         test_ydata = ydata[self.test_indexes]
36
37         return {"Xtest" :test_xdata, "Ytest" :test_ydata}
38
39     def getData(self, silent=False):
40
```

```
41          train = self.getTrainData()
42          test  = self.getTestData()
43
44          if(silent == False):
45              self.showPartitionType()
46
47          return train, test
48
49      def data_setup(self, t_set_size, n_batches, batch_t_size):
50          if(n_batches > 0 and batch_t_size == 0):
51              batch_t_size = int(t_set_size/n_batches)
52              if (batch_t_size < 1):
53                  Logger.logError("There are more batches than observed data.")
54              t_remainder = t_set_size %n_batches
55
56          elif(n_batches == 0 and batch_t_size > 0):
57              n_batches = int(t_set_size / batch_t_size)
58              if (n_batches < 1):
59                  Logger.logError("Your train size needs more data than the provided (
                        check batch size).")
60              t_remainder = t_set_size %n_batches
61
62          elif(n_batches > 0 and batch_t_size > 0):
63              Logger.logWarning("Specified number of batches and batch size,"+\
64                      "the second one takes precedence.","Partition.data_setup")
65              return self.data_setup(t_set_size, 0, batch_t_size)
66          else:
67              Logger.logError("Bad input for n_batches and batch size")
68
69          return n_batches, batch_t_size, t_remainder
70
71
72      def slice_tr_set(self, n_batches, batch_tr_size, tr_remainder, tr_indexes):
73          batches = []
74          extra, tr_base = 1, 0
75
76          while (len(tr_indexes) > tr_base):
77              if(tr_remainder > 0): tr_remainder -= 1
78              else: extra = 0
79
80              self.train_indexes = tr_indexes[tr_base:tr_base+extra+batch_tr_size]
81              tr_base += extra + batch_tr_size
82
83              batches.append(self.getTrainData())
84
85          return batches
86
87      def slice_tst_set(self, n_batches, batch_tst_size, tst_remainder, tst_indexes):
```

```
88          batches = []
89          extra, tst_base = 1, 0
90          _, ydata = self.dataset.getData()
91          batch_case = ydata[tst_indexes]
92
93          while (len(tst_indexes) > tst_base):
94              if(tst_remainder > 0): tst_remainder -= 1
95              else: extra = 0
96
97              self.test_indexes  = tst_indexes[tst_base:tst_base+extra+batch_tst_size]
98              tst_base += extra + batch_tst_size
99
100             batches.append(self.getTestData())
101
102         return batches
103
104     def generateBatches(self, tr_nbatches=0, tst_nbatches=0, tr_batch_size=0,
        tst_batch_size=0):
105         dataset_size = self.dataset.size()
106
107         #Check input and infere some data from it
108         tr_nbatches, b_tr_size, b_tr_remainder =\
109                 self.data_setup(len(self.train_indexes), tr_nbatches, tr_batch_size)
110         tst_nbatches, b_tst_size, b_tst_remainder =\
111                 self.data_setup(len(self.test_indexes), tst_nbatches, tst_batch_size)
112
113         #Generate random batches of data
114         train_indexes_old = self.train_indexes.copy()
115         test_indexes_old = self.test_indexes.copy()
116
117         tr_indexes = self.train_indexes
118         tst_indexes = self.test_indexes
119
120         Logger.logInfo("Generating "+str(tr_nbatches)+ " training batches of size ~"+
              str(b_tr_size))
121         tr = self.slice_tr_set(tr_nbatches, b_tr_size, b_tr_remainder, tr_indexes)
122         Logger.logInfo("Generating "+str(tst_nbatches)+ " testing batches of size ~"+
              str(b_tst_size))
123         tst = self.slice_tst_set(tst_nbatches, b_tst_size, b_tst_remainder,
             tst_indexes)
124
125         self.train_indexes = train_indexes_old
126         self.test_indexes = test_indexes_old
127         return tr, tst
128
129     def showPartitionType(self):
130         size = len(self.train_indexes)
131         if(size < 15):
```

```
132             Logger.logInfo("Using training partition " + str(self.train_indexes) +\
133                            " over dataset [0-"+str(self.dataset.size()-1)+"]")
134         else:
135             Logger.logInfo("Using training partition of size " + str(size) +\
136                            " over a dataset of size "+str(self.dataset.size()))
137         size = len(self.test_indexes)
138         if(size < 15):
139             Logger.logInfo("Using test partition " + str(self.test_indexes) +\
140                            " over dataset [0-"+str(self.dataset.size()-1)+"]")
141         else:
142             Logger.logInfo("Using test partition of size " + str(size) +\
143                            " over a dataset of size "+str(self.dataset.size()))
144 
145     def getSparseData(self, n):
146         if (n > len(self.train_indexes)):
147             Logger.logError("Not supported input. Using more pseudo-inputs than data.
                    ",
148                             "Partition.getSparseData")
149 
150         xdata, _ = self.dataset.getData()
151         n_train = len(self.train_indexes)
152         p = self.selector(n_train,n)
153         if(len(p) != n):
154             Logger.logError("Bad size:"+str(len(p))+"-"+str(n), "Partition.
                    getSparseData")
155         xsparse = t.empty(n, self.dataset.getXDimensionality(),
156                   dtype = xdata.dtype, device = xdata.device)
157 
158         index=0
159         for i in p:
160             xsparse[index] = xdata[self.train_indexes[i]]
161             index += 1
162             if (index >= n):
163                 break
164         xsparse.requires_grad_(True)
165         self.sparse_indexes = p
166 
167         if(index < 15):
168             Logger.logInfo("Creating a pseudo input set ["+str(p) +"]")
169         else:
170             Logger.logInfo("Creating a pseudo input set of size "+str(len(p)))
171 
172         return xsparse
173 
174 class ExplicitPartition(Partition):
175 
176     def __init__(self, dataset, indexes, selector=None):
177         size=[]
```

```
178         if(len(indexes) < 1):
179             Logger.logError("Bad number of indexes.")
180 
181         elif(type(indexes) != list):
182                 Logger.logError("Indexes must be a list of indexes.")
183         self.training_size = len(indexes)
184         self.indexes = indexes
185 
186         if(selector==None):
187             super().__init__(dataset, ExplicitPartition.batch_selector)
188         else:
189             super().__init__(dataset, selector)
190 
191     def select(self):
192         train_indexes = self.indexes
193         dataset_size = self.dataset.size()
194         test_indexes = [i for i in range(dataset_size) if i not in train_indexes]
195         return train_indexes, test_indexes
196 
197     def batch_selector(total_size, batch_size):
198         l = []
199         n_steps = int(total_size/batch_size)
200         mod = total_size%batch_size
201         #rst = number of non-necessary indexes that n_steps is going to include
202         rst = (mod//n_steps)
203         # range[0,total_size-1] In case mod == 0, total_size = n_steps*batch_size
204         # since index[d_distance*size] wont be retrieved for being outside range you
                got
205         # the exacta number of indexes. When mod > 0, and since
206         # total_size > d_distance*bsize, last index is retrievable and it needs to be
207         # removed
208         rst = rst+1 if mod%n_steps > 0 else rst
209         for i in range(0, total_size, n_steps):
210             l.append(i)
211         #remove items until len(l) == batch_size. Algorithm calculates the mean of
212         # the side items delete the items and add the mean to the list afterwards
213         index=0
214         complement=1
215         new = []
216         for i in range(rst):
217             new.append(round((l[index] + l[index+complement])/2))
218             del l[index]
219             del l[index]
220             index = (-1*index)-1
221             complement = 1 if index ==  0 else -1
222 
223         return l + new
224 
```

```
225 class RandomPartition(Partition):
226
227     def __init__(self, dataset, size, selector=None, seed = None):
228         if(size <= 0):
229             Logger.logError("Training size is not correctly defined.", "
                RandomPartition")
230         self.training_size = size
231         self.seed = t.seed() if seed == None else seed
232         t.manual_seed(self.seed)
233
234         if(selector==None):
235             super().__init__(dataset, RandomPartition.batch_selector)
236         else:
237             super().__init__(dataset, selector)
238
239     def select(self):
240         p = RandomPartition.batch_selector(self.dataset.size(),0)
241         return p[:self.training_size], p[self.training_size:]
242
243     def batch_selector(size, batch_size):
244         return t.randperm(size).tolist()
```

B.1.3. Funciones de Kernel.

Kernel

```
1  import torch as t
2  import math
3  from utilities.logger import Logger
4  #Issue, kernel hiperparapeter names are definned outside the class, but used inside
5  class Kernel:
6      def __init__(self, kernel_function, hyperp, prefix=""):
7          self.function = kernel_function
8          self.hyperp= hyperp
9          self.prefix = prefix
10         self.prepared = False
11
12     def getParams(self):
13         return self.hyperp
14
15     def getParamsAsTensors(self):
16         l = []
17         for v in self.hyperp.values():
18             l.append(v.getRawValue())
19         return l
20
21     def showParams(self):
```

```
22          row, vals = "",""
23          for k,v in self.hyperp.items():
24              row += k + ", "
25              vals += str(v.getValue().tolist()) + ", "
26          return row[:-1] +" => "+ vals[:-2]
27
28      def modify_hyperparameter(self, key, value):
29          keys = self.hyperp.keys()
30          keys = [str(k).replace(str(self),"") for k in keys]
31
32          if(key not in keys):
33              Logger.logError("Invalid key (" + key + "). Available keys: "+\
34                              str(keys), "Kernel.set_hyperparameter")
35
36          if(self.prepared == True):
37              Logger.logError("Hyperparameters prepared for opitimization can't be
                    modified",
38                              "Kernel.modify_hyperparameter")
39
40          self.set_hyperparameter(key, value)
41
42
43      def set_hyperparameter(self, key, val):
44          Logger.logError("Hyperparameters of a composed kernel cant be modified.")
45
46      def prepare_for_optim(self):
47          if(self.prepared == True):
48              Logger.logWarning("Kernel hyperparameters are already"+\
49                                "prepared for optimization.")
50              return
51          Logger.logInfo("Preparing kernel hyperparameters for optimization")
52          for k in self.hyperp.keys():
53              self.hyperp[k].optim_setup()
54          self.prepared = True
55
56
57      def process(self, x, xstar=None):
58          hyperp = self.getParams()
59
60          aux = x if xstar == None else xstar
61
62          deb_msg = "Processing kernel for:\nx = {*}\nxstar = {*}"
63          Logger.logDebug(deb_msg, [x.size(),aux.size()], "Kernel.process")
64
65          return self.function(x, xstar, hyperp, str(self))
66
67      def __add__(self, k2):
68          hyperp = {**self.hyperp, **k2.getParams()}
```

```
69          function = lambda x, xstar, hyperp, p: self.process(x,xstar) +\
70                                                 k2.process(x,xstar)
71          return Kernel(function, hyperp, str(self)+str(k2))
72
73      def __sub__(self, k2):
74          self.hyperp = {**self.hyperp, **k2.getParams()}
75          function = lambda x, xstar, hyperp, p: self.process(x,xstar) -\
76                                                 k2.process(x,xstar)
77          return Kernel(function, hyperp, str(self)+str(k2))
78
79      def __mul__(self, k2):
80          self.hyperp = {**self.hyperp, **k2.getParams()}
81          function = lambda x, xstar, hyperp, p: t.matmul(self.process(x,xstar),
82                                                          k2.process(x,xstar))
83          return Kernel(function, hyperp, str(self)+str(k2))
84
85      def test(self, x):
86          checkMatrixProperties(self.process(x))
87
88      def __str__(self):
89          return self.prefix
```

Kernel RBF

```
1  from covariances.kernel import Kernel
2  from utilities.matrix_ops import dist
3  from utilities.logger import Logger
4  from covariances.hyperparameter import LengthScale
5  from covariances.hyperparameter import Variance
6  from covariances.hyperparameter import Noise
7  import torch as t
8  import time
9
10 class RBF(Kernel):
11     def __init__(self, l_scale=None, noise=None, var=None,
12                  params = {"dtype": t.double, "device":"cpu"},
13                  jointHyp = False, kernel_function=None, free=True):
14
15         self.free = free
16         self.jointHyp = jointHyp
17         prefix = "rbf_" if(jointHyp) else ""
18         self.dev, self.typ = params["device"], params["dtype"]
19
20         hyperp = {
21                     prefix+"l_scale": LengthScale(l_scale, self.typ, self.dev, free),
22                     prefix+"noise": Noise(noise, self.typ, self.dev, free),
23                     prefix+"var": Variance(var, self.typ, self.dev, free)
24                 }
25
```

```
26          if(kernel_function):
27              super().__init__(kernel_function, hyperp)
28          else:
29              super().__init__(RBF.kernel, hyperp)
30
31      def set_hyperparameter(self, key, val):
32          ini_func = {
33                          str(self) + "l_scale": LengthScale,
34                          str(self) + "var":     Variance,
35                          str(self) + "noise":   Noise
36                      }
37
38          self.hyperp[str(self)+key] =\
39                  ini_func[key](val, self.typ, self.dev, self.free)
40
41      def kernel(x, xstar, hyperp, prefix=""):
42
43          noise           = hyperp[prefix+"noise"].getValue()
44          length_scale    = hyperp[prefix+"l_scale"].getValue()
45          variance        = hyperp[prefix+"var"].getValue()
46
47          # dividing the tensor by the lengthscale
48          x = x / length_scale
49          aux = x if t.is_tensor(xstar) == False else xstar/length_scale
50
51          #Computing similarity between points
52          dist = t.pow(t.cdist(x,aux),2)
53
54          m = variance * t.exp(-0.5 * dist)#tdist)
55
56          #Returning matrix and adding noise
57          if(t.is_tensor(xstar)): return m
58          else: return m + t.eye(m.size()[0],device=x.device) * noise
59
60
61      def __str__(self):
62          if(self.jointHyp):
63              return "rbf_"
64          else:
65              return""
```

Kernel periódico

```
1  from covariances.kernel import Kernel
2  from utilities.logger import Logger
3  from covariances.hyperparameter import LengthScale, Variance
4  from covariances.hyperparameter import Noise, Period
5  import torch as t
6  import math
```

```
7
8  class SIN(Kernel):
9      def __init__(self, l_scale=None, noise=None, var=None, period=None,
10                 params={"dtype": t.double, "device":"cpu"},
11                 jointHyp = False, kernel_function=None, free=True):
12
13         self.free = free
14         self.jointHyp = jointHyp
15         prefix = "sin_" if(jointHyp) else ""
16         self.dev, self.typ = params["device"], params["dtype"]
17
18         hyperp = {
19                     prefix+"l_scale": LengthScale(l_scale, self.typ, self.dev, free),
20                     prefix+"noise":   Noise(noise, self.typ, self.dev, free),
21                     prefix+"period":  Period(period, self.typ, self.dev, free),
22                     prefix+"s_dev":   Variance(var, self.typ, self.dev, free)
23                 }
24
25         if(kernel_function):
26             super().__init__(kernel_function, hyperp)
27         else:
28             super().__init__(SIN.kernel, hyperp)
29
30     # var* exp(-2 sin^2(pi* |x-xstar|/p)/l^2)
31     def kernel(x, xstar, hyperp, prefix=""):
32         noise           = hyperp[prefix+"noise"].getValue()
33         length_scale    = hyperp[prefix+"l_scale"].getValue()
34         var             = hyperp[prefix+"s_dev"].getValue()
35         period          = hyperp[prefix+"period"].getValue()
36
37         #dist = |x-xstar|
38         aux = xstar if t.is_tensor(xstar) else x
39
40         x_input = t.cdist(x,aux, p=1)
41         #in_sin = (pi*dist/p)
42         in_sin = (math.pi/period) * x_input
43         # -2*sin^{2}(in_sin)/l^{2}
44         exponent = -2/length_scale * t.pow(t.sin(in_sin), 2)
45
46         if(t.is_tensor(xstar)):
47             return  (var * t.exp(exponent))
48         else:
49             diag = t.mul(t.eye(in_sin.size()[0],device=aux.device), noise)
50             return (var * t.exp(exponent)) + diag
51
52     def set_hyperparameter(self, key, val):
53         ini_func = {
54                         str(self) + "l_scale": LengthScale,
```

```
55                          str(self) + "var":      Variance,
56                          str(self) + "noise":    Noise,
57                          str(self) + "period":   Period
58                      }
59
60          self.hyperp[str(self)+key] =\
61                  ini_func[key](val, self.typ, self.dev, self.free)
62
63      def __str__(self):
64          if(self.jointHyp):
65              return "sin_"
66          else:
67              return ""
```

Hiperparámetros

```
1  import torch as t
2  from data.partition import ExplicitPartition as explicit
3  from utilities.logger import Logger
4  from numbers import Number
5  class Hyperparameter:
6
7      def __init__(self, value=None, dtype=t.double, device = "cpu", free=True):
8          self.dtype = dtype
9          self.device = device
10         self.free = free
11         if(str(self) == "Hyperparameter"):
12             Logger.logError("Hyperparameter can't be instantiated as is."+\
13                             "Choose a subclass")
14         self.setValue(value)
15
16     def setValue(self, value):
17         typ, dev = self.dtype, self.device
18         if(value == None):
19             value = self.generateValue_()
20             Logger.logWarning(str(self)+" input value not found. Generating one",
21                               "Hyperparameter.setValue")
22
23         if(self.isConstrained(value) == False):
24             value = self.generateValue_()
25             Logger.logWarning(self.getValueConstrainRules()+" Generating value.",
26                               "Hyperparameter.setValue")
27
28         Logger.logDebug("Setting "+ str(self) + " to: " + str(value), [],
29                         "Hyperparameter.setValue")
30
31         v = self.transform(value)
32         self.value = t.tensor(v, dtype=typ, device=dev)
33
```

```
34      def getValue(self):
35          Logger.logDebug(str(self)+" => "+ str(self.value.item()),[] ,"Hyperparam.
                getValue")
36          return self.value
37
38      def getRawValue(self):
39          return self.value
40
41      def generateValue_(self):
42          return t.rand(1).item()
43
44      def optim_setup(self):
45          typ, dev = self.dtype, self.device
46          if(self.free):
47              self.value.requires_grad_(True)
48
49      def transform(self, value):
50          return value
51
52      def isConstrained(self, value):
53          return True
54
55      def getValueConstrainRules(self):
56          return str(self)+ " value at input doesn't comply with constrain rules."
57
58      def __str__(self):
59          return "Hyperparameter"
60
61  class RBFCommon(Hyperparameter):
62
63      def __init__(self, value=None, dtype=t.double, device = "cpu", free=True):
64          if(str(self) == "RBFCommon"):
65              Logger.logError("RBFCommon can't be instantiated")
66          else:
67              super().__init__(value, dtype, device, free)
68
69      def getValueConstrainRules(self):
70          out = str(self)+" value at input doesn't comply with constrain rules (value >
                0)."
71          return out
72
73      def isConstrained(self, value):
74          if(value > 0): return True
75          return False
76
77      def transform(self, value):
78          return t.log(t.tensor(value,dtype=self.dtype)).item()
79
```

```
80      def getValue(self):
81          v = t.exp(self.value)
82          Logger.logDebug(str(self)+" => "+ str(v.item()),[], "Hyperparam.getValue")
83          return v
84
85      def __str__(self):
86          return "RBFCommon"
87
88  class LengthScale(RBFCommon):
89
90      def __init__(self, value=None, dtype=t.double, device = "cpu", free=True,size=1):
91          self.size = size
92          if(isinstance(value, Number)):
93              super().__init__([value], dtype, device, free)
94          else:
95              super().__init__(value, dtype, device, free)
96
97      def selectLengthScale(data, max_size=20000):
98          if(data.size()[0] * data.size()[1] < max_size):
99              indices = [i for i in range(data.size()[0])]
100         else:
101             Logger.logWarning("Too much data to compute length scale, using subset")
102             indices = explicit.batch_selector(data.size()[0], max_size//data.size()
                    [1])
103
104         d = t.sort(t.pdist(data[indices],2))[0]
105         return (t.ones(data.size()[1],device=data.device) * d[d.size()[0]//2]).tolist
                ()
106
107     def getValue(self):
108         v = t.exp(self.value)
109         Logger.logDebug(str(self)+" => "+ str(v.tolist()),[], "Hyperparam.getValue")
110         return v
111
112     def transform(self, value):
113         return t.log(t.tensor(value,dtype=self.dtype)).tolist()
114
115     def generateValue_(self):
116         Logger.logWarning("Generating "+str(self.size)+ "length scale/s")
117         return [t.rand(1).item() for i in range(self.size)]
118
119     def isConstrained(self, value):
120         if(False in value > 0): return False
121         return True
122
123     def __str__(self):
124         return "LengthScale"
125
```

```
126 class Variance(RBFCommon):
127     def generateValue_(self):
128         return 0.5 + t.rand(1).item()
129 
130     def __str__(self):
131         return "Variance"
132 
133 
134 class Noise(RBFCommon):
135     def __str__(self):
136         return "Noise"
137 
138 class Period(RBFCommon):
139     def __str__(self):
140         return "Period"
141 
142 class Period(RBFCommon):
143     def __str__(self):
144         return "Period"
145 
146 class StandardParam(Hyperparameter):
147     def __init__(self, name="StandardParam", value=None, dtype=t.double, device = "
        cpu", free=True):
148         self.name=name
149         super().__init__(value, dtype, device, free)
150 
151     def __str__(self):
152         return self.name
```

B.1.4. Inferencia.

Proceso Gaussiano

```
1  from covariances.hyperparameter import LengthScale
2  from covariances.kernel import Kernel
3  from utilities.logger import Logger
4  from utilities.metric import MSE
5  import time
6  import matplotlib.pyplot as plot
7  import numpy as np
8  import torch as t
9 
10 class GaussianProcess:
11 
12     def __init__(self, partition, kernel, conf, loss = None, normalize=False):
13         self.kernel = kernel
14         self.loss = loss
15         if(loss != None):
```

```
16              self.loss.iniInferencer(self)
17          self.limited_error = False
18          self.partition = partition
19          self.train_data = None
20          self.test_data = None
21          self.training_data = {}
22          self.posterior_data = {}
23          self.trained = False
24          self.tested = False
25          self.normalize = normalize
26          self.epochs = conf["n_epochs"]
27          self.lr = conf["learning_rate"]
28          self.batch_tstsize = conf["batch_tstsize"]
29          self.prepare()
30
31      def isTrained(self):
32          return self.trained
33
34      def isTested(self):
35          if(self.isTrained() == False):
36              Logger.logError("GP can't be tested if it hasn't been trained")
37          return self.tested
38
39      def getError(self):
40          if(self.limited_error and self.loss.needsOriginal()):
41              Logger.logError("Trying to use a loss function between prediciton\
42                               and original data without the original data")
43          return self.loss
44
45      def loadNewTest(self, xtest, ytest = None):
46          self.test_data["Xtest"] = xtest
47          self.test_data["Ytest"] = ytest
48          if(ytest == None):
49              self.limited_error = True
50          self.tested = False
51
52      def prepare(self):
53          train, test = self.partition.getData()
54
55          v = LengthScale.selectLengthScale(train["Xtrain"], max_size = 100000)
56          self.kernel.modify_hyperparameter("l_scale", v)
57          #Calculate normalization params
58          if(self.normalize == True):
59              Logger.logInfo("Normalizing data")
60
61              self.x_var, self.x_mean = t.var_mean(train["Xtrain"],0)
62              if(0 in self.x_var):
63                  Logger.logError("Some feature has 0 variance (1)", "GaussianProcess.
```

```
                    prepare")
64
65              self.y_var, self.y_mean = t.var_mean(train["Ytrain"],0)
66              if(0 in self.y_var):
67                  Logger.logError("Some feature has 0 variance (2)", "GaussianProcess.
                        prepare")
68              #Normalize data
69              train["Xtrain"] = self.xnorm(train["Xtrain"])
70              test["Xtest"]   = self.xnorm(test["Xtest"])
71
72              train["Ytrain"] = self.ynorm(train["Ytrain"])
73
74          self.train_data = train
75          self.test_data = test
76
77
78      def xnorm(self, data):
79          if(self.normalize):
80              return (data - self.x_mean)/t.sqrt(self.x_var)
81          return data
82
83      def ynorm(self, data):
84          if(self.normalize):
85              return (data - self.y_mean)/t.sqrt(self.y_var)
86          return data
87
88      def mean_rescale(self, data):
89          if(self.normalize):
90              return data * t.sqrt(self.y_var) + self.y_mean
91          return data
92
93      def var_rescale(self, data):
94          if(self.normalize):
95              return data * self.y_var
96          return data
97
98      def rescale_data(self):
99          if(self.normalize == True):
100             return self.partition.getData(silent=True)
101         else:
102             return self.train_data, self.test_data
103
104     def train(self):
105         if(self.isTrained()):
106             Logger.logWarning("GP Already trained, returning cached results")
107             return self.training_data
108
109         tm = time.time()
```

```
110         x_similarity = self.kernel.process(self.train_data["Xtrain"])
111
112         #Issue t.sqrt(devuiations?)
113         self.training_data = {
114                                 "Kxx": x_similarity,
115                                 "Kxx_inv": t.inverse(x_similarity)
116                               }
117         tm = time.time() - tm
118
119         self.trained = True
120         return self.training_data, tm
121
122     def test(self, full_cov=False):
123         if(self.isTested()):
124             Logger.logWarning("GP Already tested, returning cached results")
125             return self.posterior_data
126
127         typ = self.train_data["Xtrain"].dtype
128         dev = self.train_data["Xtrain"].device
129         p_means = t.tensor([],dtype = typ, device = dev)
130         p_vars  = t.tensor([],dtype = typ, device = dev)
131
132         tm = 0
133         y = self.train_data["Ytrain"]
134         x, whole_xstar = self.train_data["Xtrain"], self.test_data["Xtest"]
135
136         if(full_cov == False):
137             for i in range(0, whole_xstar.size()[0], self.batch_tstsize):
138                 tm_loop = time.time()
139                 xstar = whole_xstar[i:i+self.batch_tstsize]
140                 Kxxstar = self.kernel.process(x, xstar)
141                 Kxstarxstar = self.kernel.process(xstar)
142
143                 Kxstarx = Kxxstar.t()
144
145                 m = self.means(Kxstarx, self.training_data["Kxx_inv"], y)
146                 v = self.variances(Kxstarxstar, Kxxstar, Kxstarx, self.training_data[
                        "Kxx_inv"])
147
148                 tm += time.time() - tm_loop
149
150                 p_means = t.cat((p_means, self.mean_rescale(m)), 0)
151                 p_vars  = t.cat((p_vars, self.var_rescale(t.diag(v))), 0)
152
153         else:
154                 tm = time.time()
155                 Kxxstar = self.kernel.process(x, whole_xstar)
156                 Kxstarxstar = self.kernel.process(whole_xstar)
```

```
157
158                 Kxstarx = Kxxstar.t()
159
160                 p_means = self.means(Kxstarx, self.training_data["Kxx_inv"], y)
161                 p_vars  = self.variances(Kxstarxstar, Kxxstar, Kxstarx, self.
                        training_data["Kxx_inv"])
162
163                 tm = time.time() - tm
164
165         #Issue t.sqrt
166         self.posterior_data = {"test_means": self.mean_rescale(p_means),
167                                "test_variances": self.var_rescale(p_vars)}
168
169         Logger.logDebug("test_means =>\n*",[p_means],"GaussianProcess.means")
170         Logger.logDebug("test_variances =>\n*",[p_vars],"GaussianProcess.variances")
171
172         self.tested = True
173
174         return self.posterior_data, tm
175
176     def means(self, Kxstarx, Kxx_inv, y, x=None, xstar=None):
177         x_mean, xstar_mean = 0, 0
178         if(t.is_tensor(x)): x_mean = x.sum()/x.size()[0]
179         if(t.is_tensor(xstar)): xstar_mean = xstar.sum()/xstar.size()[0]
180
181         return  xstar_mean + t.matmul(t.matmul(Kxstarx, Kxx_inv), (y-x_mean))[:,0]
182
183     def variances(self, Kaa, Kba, Kab, Kbb_inv):
184         return (Kaa - (t.matmul(t.matmul(Kab, Kbb_inv), Kba)))
185
186     def hyper_optim(self, xdata=None, ydata=None):
187         if(xdata == None and ydata== None):
188             xdata = self.train_data["Xtrain"]
189             ydata = self.train_data["Ytrain"]
190         elif (t.istensor(xdata) !=  t.istensor(ydata)):
191             Logger.logError("Bad input for hyper_optim", "GaussianProcess.hyper_optim
                    ")
192
193         tm = time.time()
194         metrics_time = 0
195         self.kernel.prepare_for_optim()
196
197         optimizer = t.optim.Adam(self.kernel.getParamsAsTensors(), self.lr)
198         for e in range(self.epochs):
199             Logger.logDebug("Parameters: *\n", [self.kernel.showParams()], "
                    hyperparameters")
200             optimizer.zero_grad()
201             result = self.negative_marginal_likelihood(xdata,ydata)
```

```
202             result.backward()
203             Logger.logDebug("Marginal likelihood: *\n", [result.item()],"negativeML")
204 
205             metrics_time += self.measurement()
206             optimizer.step()
207             Logger.progress(e, self.epochs,"epoch")
208 
209         Logger.progress(self.epochs, self.epochs,"epoch")
210 
211         return time.time() - tm
212 
213     def negative_marginal_likelihood(self,xdata, ydata):
214         k = self.kernel.process(xdata)
215         k_inverse = k.inverse()
216         ydatat = ydata.t()
217 
218         complexity = t.logdet(k)                              # log |Ky|
219         constant = k.size()[0] * t.log(t.tensor(2*np.pi))     # N * log |2pi|
220         fit = t.matmul(t.matmul(ydatat, k_inverse), ydata)    # y * K^{-1} * y
221 
222         return 0.5 * (fit + complexity + constant)
223 
224     def measurement(self):
225         if(self.loss == None):
226             return 0
227 
228         tmp=0
229         self.trained = True
230         tsted = self.tested
231         self.tested = False
232         with t.no_grad():
233             tmp = time.time()
234             self.loss.compute_by()
235             tmp = time.time() - tmp
236         Logger.unbuffer_debug()
237         self.tested = tsted
238 
239         return tmp
240     def get_posterior_data(self):
241         if(self.isTested()==False):
242             Logger.logError("GP needs to be tested before plotting")
243         return (self.posterior_data["test_means"].detach(),
244                 self.posterior_data["test_variances"].detach())
245 
246     def get_repr_data(self):
247         if(self.isTested()==False):
248             Logger.logError("GP needs to be tested before plotting")
249         train, test = self.rescale_data()
```

```
250         tr_order = np.argsort(self.partition.train_indexes)
251         ts_order = np.argsort(self.partition.test_indexes)
252 
253         if(self.partition.dataset.getXDimensionality() > 1):
254             x_tr = np.array(self.partition.train_indexes)[tr_order]
255             x_ts = np.array(self.partition.test_indexes)[ts_order]
256         else:
257             x_tr = np.array(train["Xtrain"][:,0].cpu().detach())[tr_order]
258             x_ts  = np.array(test["Xtest"][:,0].cpu().detach())[ts_order]
259 
260         y_tr = np.array(train["Ytrain"][:,0].cpu().detach())[tr_order]
261         y_ts  = np.array(test["Ytest"][:,0].cpu().detach())[ts_order]
262         p_mean  = np.array(self.posterior_data["test_means"].cpu().detach())[ts_order
                ]
263         p_vars  = np.array(2*t.sqrt(self.posterior_data["test_variances"].cpu().
                detach()))[ts_order]
264         return (x_tr, y_tr, x_ts, y_ts, p_mean, p_vars)
265 
266     def plot_posterior(self, path, dpi):
267         xtrain, ytrain, xtest, ytest, means, deviations =  self.get_repr_data()
268         if(self.train_data["Xtrain"].size()[1] > 1):
269             Logger.logWarning("Posterior plot only supports 1-feature data")
270             return
271 
272         f = plot.figure()
273         plot.title("Observations and posterior")
```

Proceso Gaussiano Aproximado

```
1  from inference.gaussian import GaussianProcess
2  from torch.distributions import MultivariateNormal
3  from torch.distributions.kl import kl_divergence
4  from covariances.rbfKernel import RBF
5  from utilities.matrix_ops import positive_definite_diag, checkMatrixProperties
6  from utilities.logger import Logger
7  from utilities.debug import getGraph
8  import matplotlib.pyplot as plot
9  import time
10 import torch as t
11 import numpy as np
12 
13 
14 class SparseGaussianProcess(GaussianProcess):
15 
16     def __init__(self, partition, kernel, conf, loss_f = None, normalize=False):
17         self.batch_trsize = conf["batch_trsize"]
18         self.n_batches = conf["n_batches"]
19         super().__init__(partition, kernel,conf, loss_f, normalize)
20         self.posterior_data = {}
```

```
21
22          # Inducing set initialization
23          with t.no_grad():
24              s_data = self.partition.getSparseData(conf["u_size"])
25              self.pseudo_inputs_ini = s_data.detach().clone()
26              self.pseudo_inputs = self.xnorm(s_data)
27          self.pseudo_inputs.requires_grad_(True)
28
29          # Inducing set means
30          self.mu = t.zeros([conf["u_size"],1], dtype=self.train_data["Xtrain"].dtype,
31                    device=self.train_data["Xtrain"].device,requires_grad=True)
32
33          #Initialize L to some positive definite values and saving the log to always
                recover a positive definite
34          with t.no_grad():
35              chol_Kuu = t.cholesky(self.kernel.process(self.get_pseudo_inputs()))
36              self.log_L = chol_Kuu - t.diag(t.diag(chol_Kuu)) + t.diag(t.log(t.diag(
                    chol_Kuu)))
37          self.log_L.requires_grad_(True)
38
39          self.sgp_var = t.tensor(-5, dtype=self.mu.dtype, device = self.mu.device,
                requires_grad=True)
40
41      def getL(self):
42          r = range(self.log_L.size()[0])
43          L = self.log_L.clone()
44          L[r,r] = t.exp(L[r,r])
45          return L
46
47      def getMu(self):
48          return self.mu
49
50      def getS(self):
51          L = self.getL()
52          return t.matmul(L, L.t())
53
54      def get_pseudo_inputs(self):
55          return self.pseudo_inputs
56
57      def get_sgp_params(self):
58          return [self.pseudo_inputs, self.log_L, self.mu, self.sgp_var]
59
60      def testLogLikelihood(self):
61          test_means = self.posterior_data["test_means"]
62          test_vars  = self.posterior_data["test_variances"]
63          ystar = self.test_data["Ytest"][:,0]
64
65          log_like = self.logLikelihood(test_means,test_vars, ystar)
```

```
66          return log_like.sum()/test_means.size()[0]
67
68      def logLikelihood(self, p_means, p_vars, data):
69          return -0.5 * t.log(2 * np.pi * p_vars) - 0.5 * t.pow(data-p_means,2)/p_vars
70
71
72      def compute_optimizer(self,optimizer, result):
73          optimizer.zero_grad()
74          result.backward()
75          optimizer.step()
76
77      def train(self, errorMeasurement=False):
78          if(self.isTrained()):
79              Logger.logWarning("GP already trained")
80              return None
81
82          self.likelihood_means = []
83          training_time, metrics_time = 0, 0
84
85          #Configure optimizer
86          self.kernel.prepare_for_optim()
87          kern_params = self.kernel.getParamsAsTensors()
88          sgp_params = self.get_sgp_params()
89          optimizer = t.optim.Adam(kern_params + sgp_params, lr = self.lr)
90
91          self.train_batches, self.test_batches =\
92              self.partition.generateBatches(self.n_batches, 0, self.batch_trsize, self
                    .batch_tstsize)
93          nbatches = len(self.train_batches)
94
95          for e in range(self.epochs):
96              i=0
97              self.likelihood = 0
98              Logger.progress(e,self.epochs, "epochs")
99              for train in self.train_batches:
100                 Logger.progress(i, nbatches, "sparse_minibatch")
101
102                 time_1 = time.time()
103
104                 x, y = self.xnorm(train["Xtrain"]), self.ynorm(train["Ytrain"])
105                 m, v = self.sparse_optimization(x)
106                 result = self.elbo(y, m, v, self.Kuu, x.size()[0])
107
108                 time_2 = time.time()
109                 self.likelihood += result.item()
110                 metrics_time += self.measurement()
111
112                 time_3 = time.time()
```

```
113                 self.compute_optimizer(optimizer, result)
114                 tm = (time.time() - time_3) + (time_2-time_1)
115 
116                 Logger.logDebug("Iteration time = *",[tm],"iter_time")
117                 training_time += tm
118 
119                 i += 1
120 
121             Logger.progress(nbatches, nbatches,"sparse_minibatch")
122             self.likelihood_means.append(self.likelihood/nbatches)
123 
124         Logger.progress(self.epochs, self.epochs, "epochs")
125 
126         self.trained = True
127         return self.training_data, training_time, metrics_time
128 
129     def test(self):
130         if(self.isTrained() != True):
131             Logger.logError("Trying to predict without fitting first.")
132         if(self.isTested()): return self.posterior_data, 0
133 
134         test_means = t.tensor([],dtype = self.Kuu.dtype, device = self.Kuu.device)
135         test_vars  = t.tensor([],dtype = self.Kuu.dtype, device = self.Kuu.device)
136 
137         ts_time = time.time()
138         for t_data in self.test_batches:
139             xstar = self.xnorm(t_data["Xtest"])
140             Kxstaru = self.kernel.process(xstar, self.get_pseudo_inputs())
141 
142             m = self.means(Kxstaru, self.Kuu_chol, self.getMu())
143             v = self.variances(Kxstaru, self.Kuu_chol, self.getL())
144 
145             test_means = t.cat((test_means, self.mean_rescale(m)), 0)
146             test_vars  = t.cat((test_vars, self.var_rescale(v)), 0)
147 
148         ts_time = time.time() - ts_time
149         self.posterior_data["test_means"] = test_means
150         self.posterior_data["test_variances"] = test_vars
151         self.tested=True
152 
153         return self.posterior_data, ts_time
154 
155 
156     def sparse_optimization(self, x):
157 
158         u = self.get_pseudo_inputs()
159         self.Kuu = self.kernel.process(u)
160         self.Kuu_chol = t.cholesky(self.Kuu)
```

```
161
162         Kxu = self.kernel.process(x, u)
163
164         f_means = self.means(Kxu, self.Kuu_chol, self.getMu())
165         f_vars = self.variances(Kxu, self.Kuu_chol, self.getL())
166
167         self.posterior_data["train_means"] = f_means
168         self.posterior_data["train_vars"]  = f_vars
169
170         return f_means, f_vars
171
172     def elbo(self, y, m, v, Kuu, norm_factor):
173         var = t.exp(self.sgp_var)
174         mu = self.getMu()
175
176         L = self.getL()
177
178         E_q = -0.5 * t.log(2 * np.pi * var)\
179               -0.5 * t.div(t.pow(y[:,0],2), var)\
180               -0.5 * t.div(((-2 * t.mul(y[:,0], m)) + v + t.pow(m,2)),var)
181
182         E_q = E_q.sum()
183         kl = self.kl_torch(Kuu, mu, L)
184
185         Logger.logDebug("Eq => *", [E_q.item()], "SparseGaussianProcess.Eq")
186         Logger.logDebug("kl => *", [kl.item()],  "SparseGaussianProcess.kl")
187
188         return -(E_q*norm_factor - kl)
189
190     def kl_torch(self, Kuu, mu, L):
191         p = MultivariateNormal(t.zeros(mu.size()[0],device=Kuu.device), Kuu)
192         q = MultivariateNormal(mu[:,0], scale_tril=L)
193         return kl_divergence(q,p).sum()/Kuu.size()[0]
194
195     def means(self, Kab, Kbb, m):
196         return t.matmul(Kab,t.cholesky_solve(m,Kbb))[:,0]
197
198     def variances(self, Kab, Kbb, L):
199         var = self.kernel.getParams()["var"].getValue() #Kxx diag
200         var2 = t.exp(self.sgp_var)
201
202         Kba = Kab.t()
203         Kbbinv_Kba = t.cholesky_solve(Kba,Kbb)
204         LtKbbinv_Kba = t.matmul(L.t(), Kbbinv_Kba)
205         t1 = (Kbbinv_Kba * Kba).sum(0)
206         t2 = (LtKbbinv_Kba * LtKbbinv_Kba).sum(0)
207
208         return  var + var2 - t1 + t2
```

```
209
210     def plot_posterior(self, path, dpi):
211         xtrain, ytrain, xtest, ytest, means, deviations =  self.get_repr_data()
212         if(self.train_data["Xtrain"].size()[1] > 1):
213             Logger.logWarning("Posterior plot only supports 1-feature data")
214             return
215
216         f = plot.figure()
217         plot.title("Observations and posterior")
218         p_input = self.get_pseudo_inputs().cpu()
219         #means
220         plot.plot(xtest, means , 'g-', label='mean')
221         #observations
222         plot.plot(xtrain,ytrain,'r.', label='observations')
223         plot.plot(xtest, ytest,'m*', label='targets')
224
225         #standard deviations
226         p_dev_neg = means - deviations
227         p_dev_plu = means + deviations
228         plot.fill_between(xtest, p_dev_neg, p_dev_plu, color="g",edgecolor=None,alpha
                =0.2, label="covariance")
229         #Pseudo inputs
230         bot, top = plot.ylim()
231         plot.plot(p_input.detach().numpy(),t.empty(p_input.size()[0]).fill_(bot).
                detach().numpy(),'m+', label='pseudo_inputs')
232         plot.plot(self.pseudo_inputs_ini.cpu().numpy(),t.empty(p_input.size()[0]).
                fill_(top).cpu().numpy(),'y+', label='pseudo_inputs_ini')
```

B.1.5. Modulos auxiliares.

Métricas de calidad de ajuste.

```
1  import torch as t
2  import time
3  import matplotlib.pyplot as plot
4  from torch.nn.functional import mse_loss
5  from utilities.logger import Logger
6
7  #
8  #   Important when using class Errors to compute_by_time(), beware that the
9  #   timestamp is common for the errors. In other words, processing to genereate
10 #   de data wont be taken into account. If you want to take it into account,
11 #   process it in your code and not in the getParams() subclass method.
12 #
13
14 class Metric:
15     def __init__(self, metric_fun, timer=1, mode="time"):
16         self.metric_fun = metric_fun
```

```
17          self.time_init = 0
18          self.mode = mode
19          self.log_timestamps = []
20          self.log_metric = []
21          self.inferencer = None
22          self.initialized = False
23          self.timer = timer  # counts start at 1, but one iteration will be done at 0
24          self.used_timer = 0
25
26          if(timer < 0):
27              Logger.logWarning("Trying to measure ["+str(self)+"] but cant be"+\
28                                " measured every ("+str(timer)+") batches")
29          else:
30              Logger.logInfo("Measurement of [*] will be done every (*) batches",
31                             "", [str(self),str(timer)])
32
33      def getComputationInputs():
34          pass
35
36      def iniInferencer(self, inferencer):
37          self.inferencer = inferencer
38
39      def initialize_time(self):
40          self.time_init = time.time()
41          self.initialized=True
42
43      def compute_by_time(self):
44          if(self.initialized == False):
45              self.initialize_time()
46          if (self.checkTime() == False): return
47          res = self.compute(self.getComputationInputs())
48          if(res != None):
49              self.log_metric.append(res)
50              self.log_timestamps.append(time.time()-self.time_init)
51
52      def compute_by_batch(self):
53          res = None
54          if (self.checkTime() == True):
55              res = self.compute(self.getComputationInputs())
56          if(res != None):
57              self.log_metric.append(res)
58              self.log_timestamps.append(self.time_init)
59          self.time_init +=1
60
61      def compute_by(self):
62          if self.mode == "time":
63              self.compute_by_time()
64          elif mode == "batch":
```

```
65              self.compute_by_batch()
66          else:
67              Logger.logError("Bad input for metric/s mode. Use: 'time' or 'batch'.")
68
69
70      def checkInput(self, data):
71          if(data[0].size() != data[0].size()):
72              Logger.logError("Bad input at metric computation")
73          return True
74
75      def checkTime(self):
76          sol = False
77          if(self.used_timer %self.timer == 0): sol = True
78          self.used_timer +=1
79          return sol
80
81      def compute(self, data):
82          if(self.checkInput(data) == None): return None
83          res =  self.metric_fun(data)
84          Logger.logDebug(str(self)+": "+str(res), [] ,"Metric.compute")
85          #self.used_timer= 0
86
87          return res
88
89      def plot(self, path, dpi):
90          f = plot.figure()
91          plot.ylabel('error')
92          if self.mode == "time":
93              plot.xlabel('time (s)')
94          if self.mode == "batch":
95              plot.xlabel('iterations')
96          #plot.title(str(self))
97          plot.plot(self.log_timestamps,self.log_metric, "b-", label=str(self))
98
99          legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,1.1), ncol=3)
100         f.savefig(path, dpi=dpi, bbox_extra_artists = [legend],bbox_inches="tight")
101
102
103     def __str__(self):
104         return "Metric"
105
106 class Metrics(Metric):
107     def __init__(self, *metrics, mode="time", jointPlot=False):
108         self.time_init = 0
109         self.metrics = metrics
110         self.log_timestamps = {}
111         self.log_metric = {}
112         self.mode = mode
```

```
113         self.jointPlot = jointPlot
114         for e in self.metrics:
115             self.log_timestamps[str(e)] = []
116             self.log_metric[str(e)] = []
117 
118     def iniInferencer(self, inferencer):
119         for e in self.metrics:
120             e.iniInferencer(inferencer)
121 
122 
123     def compute_by_time(self):
124         if(self.time_init == 0):
125             self.time_init = time.time()
126         tstamp = time.time()
127         for e in self.metrics:
128             res=None
129             if (e.checkTime() == True):
130                 res = e.compute(e.getComputationInputs())
131             if(res != None):
132                 self.log_metric[str(e)].append(res)
133                 self.log_timestamps[str(e)].append(tstamp-self.time_init)
134 
135     def compute_by_batch(self):
136         for e in self.metrics:
137             res=None
138             if (e.checkTime() == True):
139                 res = e.compute(e.getComputationInputs())
140             if(res != None):
141                 self.log_metric[str(e)].append(res)
142                 self.log_timestamps[str(e)].append(self.time_init)
143         self.time_init +=1
144 
145     def plot(self, path, dpi):
146         colors = ["b","g", "r","y", "m"]
147         i=0
148 
149         if(self.jointPlot):
150             f = plot.figure()
151             for e in self.metrics:
152                 #plot.title(str(self))
153                 Logger.logDebug(str(self.log_timestamps[str(e)])+"=>"+str(self.
                        log_metric[str(e)]),[],"Metrics.plot")
154                 plot.plot(self.log_timestamps[str(e)],self.log_metric[str(e)], colors
                        [i%len(colors)]+"-", label=str(e))
155                 legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,1.1),
                        ncol=3)
156                 i += 1
157 
```

```
158             f.savefig(path+str(e), dpi=dpi, bbox_extra_artists = [legend],bbox_inches
                    ="tight")
159
160         else:
161             for e in self.metrics:
162                 f = plot.figure()
163                 plot.ylabel('value')
164                 if self.mode == "time":
165                     plot.xlabel('time (s)')
166                 if self.mode == "batch":
167                     plot.xlabel('iterations')
168                 #plot.title(str(e))
169                 plot.xscale('log')
170                 Logger.logDebug(str(self.log_timestamps[str(e)][-10:]) +"=>"+str(self
                        .log_metric[str(e)][-10:]),[],"Metrics.plot")
171                 plot.plot(self.log_timestamps[str(e)],self.log_metric[str(e)], colors
                        [i%len(colors)]+"-", label=str(e))
172                 legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,1.1),
                        ncol=3)
173                 f.savefig(path+str(e), dpi=dpi, bbox_extra_artists = [legend],
                        bbox_inches="tight")
174
175                 i += 1
176
177     def __str__(self):
178         return "Metrics"
179
180 class MSE(Metric):
181     def __init__(self, timer=1, mode="time"):
182         super().__init__(MSE.calculate,timer)
183
184     def __str__(self):
185         return "MeanSquaredError"
186
187     def calculate(data, reduc='mean'):
188         predictive, target = data
189         res = mse_loss(predictive, target, reduction=reduc).item()
190         Logger.logDebug("MSE = *",[res],"MSE", buffered=True)
191         return res
192
193     def getComputationInputs(self):
194         self.inferencer.test()
195         data = (self.inferencer.posterior_data["test_means"],
196                 self.inferencer.test_data["Ytest"][:,0])
197         return data
198
199 class RMSE(Metric):
200     def __init__(self, timer=1, mode="time"):
```

```
201         super().__init__(RMSE.calculate,timer)
202
203     def __str__(self):
204         return "RootedMeanSquaredError"
205
206     def calculate(data, reduc='mean'):
207         predictive, target = data
208         res = t.sqrt(mse_loss(predictive, target, reduction=reduc)).item()
209         Logger.logDebug("RMSE = *",[res],"RMSE", buffered=True)
210         return res
211
212     def getComputationInputs(self):
213         self.inferencer.test()
214         data = (self.inferencer.posterior_data["test_means"],
215                 self.inferencer.test_data["Ytest"][:,0])
216         return data
217
218 class SMSE(Metric):
219     def __init__(self, timer=1, mode="time"):
220         super().__init__(SMSE.calculate,timer)
221
222     def __str__(self):
223         return "StandardMeanSquaredError"
224
225     def calculate(data, reduc='mean'):
226         predictive, v, target = data
227         res = ((t.pow(target - predictive, 2)/v).sum()/target.size()[0]).item()
228         Logger.logDebug("SMSE = *",[res],"SMSE", buffered=True)
229         return res
230
231     def getComputationInputs(self):
232         self.inferencer.test()
233         data = (self.inferencer.posterior_data["test_means"],
234                 self.inferencer.posterior_data["test_variances"],
235                 self.inferencer.test_data["Ytest"][:,0])
236         return data
237
238 class RSMSE(Metric):
239     def __init__(self, timer=1, mode="time"):
240         super().__init__(RSMSE.calculate,timer)
241
242     def __str__(self):
243         return "RootedStandardMeanSquaredError"
244
245     def calculate(data, reduc='mean'):
246         predictive, v, target = data
247         res = (t.sqrt((t.pow(target - predictive, 2)/v).sum()/target.size()[0])).item
                ()
```

```
248         Logger.logDebug("RSMSE = *",[res],"RSMSE", buffered=True)
249         return res
250 
251     def getComputationInputs(self):
252         self.inferencer.test()
253         data = (self.inferencer.posterior_data["test_means"],
254                 self.inferencer.posterior_data["test_variances"],
255                 self.inferencer.test_data["Ytest"][:,0])
256         return data
257 
258 class TestLogLikelihood(Metric):
259     def __init__(self,timer=1, mode="time"):
260         super().__init__(TestLogLikelihood.calculate,timer)
261 
262     def __str__(self):
263         return "NegativeTestLogLikelihood"
264 
265     def calculate(log_likelihood):
266         Logger.logDebug("NTst LL = *",[log_likelihood.item()],"test_loglikelihood",
                buffered=True)
267         return log_likelihood
268 
269     def getComputationInputs(self):
270         self.inferencer.test()
271         return -self.inferencer.testLogLikelihood()
272 
273     def checkInput(self, data):
274         return True
275 
276 class MLElbo(Metric):
277     def __init__(self,timer=1, mode="time"):
278         super().__init__(MLElbo.calculate,timer)
279 
280     def __str__(self):
281         return "MLElbo"
282 
283     def calculate(data):
284         Logger.logDebug("ELBO = *", [-data], "elbo", buffered=True)
285         return -data
286 
287     def getComputationInputs(self):
288         return self.inferencer.likelihood/(self.timer)
289 
290     def iniInferencer(self, inferencer):
291         self.inferencer = inferencer
292 
293     def checkInput(self, data):
294         if(data == None): return None
```

```
295         return True
```

Métricas de rendimiento.

```
1  from utilities.logger import Logger
2  import timeit
3  import time
4
5  def incr_x_size(params, step, i):
6      params["xn_data"] = params["xn_data"] + step
7      return step + i
8
9  def time_measurement(func, params, incr_fun=incr_x_size):
10     min_size, max_size, step = params["incr_start"], params["incr_max"], params["step
           "]
11     l = Logger.current_lvl
12     Logger.logInfo("Restricting logs to level 3 (status bars)")
13     if(l > 3): Logger.setLogLevel(3)
14
15     if(params["gpu"]):
16         Logger.logInfo("Starting gpu measurement")
17         results = cpu_measurement(func, params, min_size, max_size, step, incr_fun)
18
19     else:
20         Logger.logInfo("Starting cpu measurement")
21         results =  cpu_measurement(func, params, min_size, max_size, step, incr_fun)
22
23     Logger.setLogLevel(l)
24     return results
25
26 def save_time_data(params, func, iter_results, time_results, f_results,i):
27
28     code_time = func(params, needPlot = False)
29     iter_results.append(i)
30     vals = [i for i in code_time.values()]
31
32     time_results.append(code_time)
33     f_results.append(str(i) + ", " + str(vals)[1:-1] + "," + str(params["xn_data"])
           +\
34                      "," + str(params["xstarn_data"]) + "\n")
35
36 def cpu_measurement(func, params,min_size, max_size, step, incr_fun):
37     iter_results = []
38     time_results = []
39     f_results = []
40
41     i=min_size
42     while(i < max_size):
43         Logger.progress(i, max_size, "input size")
```

```
44          i = incr_fun(params, step, i)
45          save_time_data(params, func, iter_results, time_results, f_results, i)
46      with open(params["performance_dir"] + params["test_name"] + ".txt", 'a') as f:
47          f.writelines(f_results)
48
49      Logger.progress(max_size, max_size, "input size")
50      return iter_results, time_results
51
52  def time_improvement(t1, t2):
53      return 100 - (t2*100/t1)
54
55  def used_time(fun):
56      tm = time.time()
57      out = fun()
58      tm = time.time() -tm
59
60      return out, tm
61
62  def incr_tr_size(params, step, i):
63      if(i < params["u_data"]):
64          Logger.logWarning("Start size can't be lower than inducing set size.","
                incr_tr_size")
65          Logger.logWarning("Initializing start size to inducing set size.", "
                incr_tr_size")
66          return params["u_data"]
67
68      params["batch_trsize"] = params["batch_trsize"] + step
69      return step + i
```

Monitorizando la ejecución.

```
1   class Logger:
2       #Log level conf
3       levels = [ 0, 1, 2, 3, 4, 5]
4       modes = ["all", "desc", "fixed"]
5       mode = "desc" # 0 no debug, 1 errors & warnings, 2 all
6       current_lvl = 1
7
8       colors = {
9               "Warning":'\033[1;33m',
10              "Error":  '\033[1;31m',
11              "Info":   '\033[1;36m',
12              "Status": '\033[1;32m',
13              "Debug":  '\033[1;95m',
14              "end":    '\033[0m'
15          }
16      lvl_relation = {
17              "Error":   1,
18              "Warning": 2,
```

```
19              "Status":  3,
20              "Info":    4,
21              "Debug":   5
22          }
23
24      mode_choices = {
25              "all": lambda x,y: x <= y,
26              "desc": lambda x,y: x <= y,
27              "fixed": lambda x,y: x == y
28          }
29
30
31      #history
32      progress_history = {}
33      warnings_history = {}
34      debug_buffer = ""
35
36      #Debug
37      debug_modes = ["text", "file"] #ToDo Graph
38      debug_mask = []
39      debug_mode = "text"
40
41      dir_path = ""
42
43      def setLogConf(conf):
44          if "log_level" in conf:  Logger.setLogLevel(conf["log_level"])
45          if "log_mode"  in conf:  Logger.setLogMode(conf["log_mode"])
46          if "log_dmode" in conf:  Logger.setDebugMode(conf["log_dmode"])
47          if "log_dmask" in conf:  Logger.setDebugMask(conf["log_dmask"])
48          if "log_dir"   in conf:  Logger.setLogDir(conf["log_dir"])
49          if "colorless" in conf:
50              if(conf["colorless"]): Logger.setColorless()
51
52      def setLogDir(dir_path):
53          Logger.dir_path = dir_path
54
55      def getLogLevel():
56          return Logger.current_lvl
57
58      def setLogLevel(lvl):
59          if(lvl in Logger.levels):
60              Logger.current_lvl=lvl
61              Logger.logInfo("Setting log level to " + str(lvl))
62          else:
63              Logger.current_lvl=1
64              Logger.logError("Not a valid level","Logger.setLogLevel")
65
66      def setLogMode(mode):
```

```
67          if(mode in Logger.modes):
68              Logger.logInfo("Setting log mode to: "+ mode)
69              Logger.mode = mode
70          else:
71              Logger.logError("Not a valid mode", "Logger.setLogMode")
72
73      def setDebugMode(mode):
74          if(mode in Logger.debug_modes):
75              Logger.debug_mode=mode
76              Logger.logInfo("Setting debug mode to: "+ mode)
77              if(mode == "file" and Logger.colors["Debug"] != ''):
78                  msg = "Recommended use of --colorless when using file debug mode."
79                  Logger.logWarning(msg)
80          else:
81              Logger.logError("Not a valid mode", "Logger.setDebugMode")
82
83      def setDebugMask(mask):
84          mask = mask.replace(" ", "")
85          mask = mask.split(",")
86          if(len(mask)<50):
87              mask = [i for i in mask if i != ""]
88              Logger.debug_mask=mask
89              Logger.logInfo("Setting debug mask to: "+ str(mask))
90          else:
91              Logger.logError("Not a valid mask", "Logger.setDebugMode")
92
93      def setColorless():
94          Logger.colors = {
95                  "Warning":'',
96                  "Error":  '',
97                  "Info":   '',
98                  "Status": '',
99                  "Debug":  '',
100                 "end":    ''
101             }
102
103
104     def canBeLogged(color):
105         return Logger.mode_choices[Logger.mode](Logger.lvl_relation[color], Logger.
                current_lvl)
106
107     def progress(completion, total, name):
108         if(Logger.canBeLogged("Status") == False):
109             return
110         if(total > 500):
111             Logger.progress_history[name]="["+str(completion)+"/"+str(total)+"] "
112         else:
113             Logger.progress_history[name]=str(int(100*completion/total))+"%  "
```

```
114         color = Logger.colors["Warning"]
115         final = ""
116         cont = len(Logger.progress_history.keys())
117         line_end= "\t"
118 
119         for k,v in Logger.progress_history.items():
120             if(v == "100 %   " or v == "["+str(total)+"/"+str(total)+"] "):
121                 color = Logger.colors["Status"]
122             line_end = "".join([" " for i in range(len(str(total))*2 + 7 - len(v))])
123             if(cont == 1): line_end +=""
124 
125             final += Logger.colors["Info"] + "Status ("+ k +"): " +\
126                     color + v + Logger.colors["end"] + line_end
127 
128             cont = cont - 1
129 
130 
131         if(completion/total == 1):
132             del Logger.progress_history[name]
133             line_end = "\n" if Logger.progress_history == {} else ""
134             print("\r" + final, end = line_end)
135         else:
136             print("\r" + final, end="")
137 
138 
139     def log(color, message, origin="", output=None):
140         origin = " (" + origin + ")" if origin != "" else origin
141         if(output):
142             pass
143         elif (Logger.canBeLogged(color)):
144             print(Logger.colors[color] + color + origin +  ": " + message + Logger.
                    colors["end"])
145 
146         if(color == "Error"):
147             exit()
148         return
149 
150 
151     def logDebug(message, values=[], origin="", buffered=False):
152         if(Logger.debug_mask != [] and origin not in Logger.debug_mask):
153             return
154         message = Logger.replace(message,values)
155         if(buffered == True):
156             l = len(message)
157             padding= ""
158             if(l > 30):
159                 Logger.logWarning("Logger padding overflow "+message + str(l))
160             else:
```

```
161                 padding = "".join([" " for i in range(30-l)])
162
163             Logger.debug_buffer = Logger.debug_buffer + " " + message + padding
164
165         elif(Logger.debug_mode == "text"):
166             Logger.log("Debug", message, origin)
167         elif(Logger.debug_mode == "graph"):
168             Logger.logWarning("Debug mode 'graph' not supported, change configuration
                ", "Logger.logDebug")
169
170         elif(Logger.debug_mode == "file"):
171             with open(Logger.dir_path +"_debug"+ ".log", 'a') as f:
172                 f.writelines(message+"\n")
173         return
174
175     def unbuffer_debug():
176         if(Logger.debug_buffer == ""): return
177         Logger.logDebug(Logger.debug_buffer ,[],"buffered")
178         Logger.debug_buffer=""
179
180     def logInfo(message, origin="", values=[], output=None):
181         message = Logger.replace(message,values,
182                   color=Logger.colors["Status"], base_color = Logger.colors["Info"])
183         Logger.log("Info", message, origin, output)
184
185     def logError(message, origin="", output=None):
186         Logger.log("Error", message, origin, output)
187
188     def logWarning(message, origin="", output=None):
189         key=""
190         if(origin == ""): key=message
191         else: key=origin
192         try:
193             Logger.warnings_history[key] += 1
194         except:
195             Logger.warnings_history[key] = 1
196
197         Logger.log("Warning", message, origin, output)
198
199     def warningSummary():
200         total = 0
201         output = ""
202         for k,v in Logger.warnings_history.items():
203             output += "\n   => " + k +": " + str(v)
204             total += v
205         print(Logger.colors["Warning"] +\
206                 "}_____________ Warning Summary _____________{\n"+\
207                 "   => Number of warnigs = " + str(total) + output +\
```

```
208                 Logger.colors["end"])
209
210     def replace(old, new, color="", base_color=""):
211         for i in range(len(new)):
212             old = old.replace("*", color + str(new[i]) + base_color,1)
213         return old
214
215     def err(message):
216         return Logger.colors["Error"] + message + Logger.colors["end"]
```

Lectura de la entrada del programa.

```
1  import json
2  import argparse
3  import torch as t
4  from utilities.logger import Logger
5
6  def repack(params):
7      return {
8              "u_size":           params["u_data"],
9              "learning_rate":    params["learning_rate"],
10             "n_epochs":         params["epochs"],
11             "n_batches":        params["n_batches"],
12             "batch_trsize":     params["batch_trsize"],
13             "batch_tstsize":    params["batch_tstsize"]
14             }
15
16 def conf_load(args, conf_file='input/conf/default_conf.json'):
17     overrided_params = conf_override(args)
18     if('conf_file' in overrided_params):
19         params = conf_read(overrided_params['conf_file'])
20     else:
21         Logger.logWarning("Using default conf file")
22         params = conf_read(conf_file)
23
24     for k,v in overrided_params.items():
25         params[k] = v
26
27     todel = []
28     for k,v in params.items():
29         if(v == "None"):
30             params[k]=None
31         elif(v == "Empty"):
32             todel.append(k)
33         elif(v == "true"):
34             params[k] = True
35         elif(v == "false"):
36             params[k] = False
37     for k in todel:
```

```
38          del params[k]
39
40
41      if(params["dtype"] == "float"):
42          params["dtype"] = t.float
43
44      elif(params["dtype"] == "double"):
45          params["dtype"] = t.double
46
47      else:
48          Logger.logError("Non supported dtype. Available dtypes: float|double")
49
50      return params
51
52  def conf_override(args):
53      parser = argparse.ArgumentParser()
54
55      #Program test and plot options
56      parser.add_argument('--pdir', '--performance-dir', dest='performance_dir',
57                          type=str, help='Relative path to performance file directory'
                                )
58      parser.add_argument('--ldir', '--log-dir', dest='log_dir', type=str,
59                          help='Relative path to log file directory')
60      parser.add_argument('--dfile', '--dataset-file', dest='dataset_file', type=str,
61                          help='Relative path to dataset file directory')
62      parser.add_argument('--idir', '--image-dir', dest='image_dir', type=str,
63                          help='Relative path to image file directory')
64      parser.add_argument('--seed', dest='seed', type=float,
65                          help='Default seed for random generation')
66      parser.add_argument('--colorless', dest='colorless', nargs='?',type=bool,
67                         const=True,default=False, help='No color in output')
68
69      #tests
70      parser.add_argument('-t', '--tests', dest='tests', type=str,
71                  help='list of tests to be executed (ex: 1,2,3). Only numbers and
                        comas')
72      parser.add_argument('-cfile', '--conf-file', dest='conf_file', type=str,
73                   help='Input configuration file in tests/ dir. Do not use relative
                         path')
74
75      #Set sizes
76      parser.add_argument('--x-size', type=int, dest='xn_data',
77                          help='Number of x data to be generated')
78      parser.add_argument('--n-size', dest='xstarn_data', type=int,
79                          help='Number of xstar data to be generated')
80      parser.add_argument('--u-size', dest='u_data', type=int,
81                          help='Number of pseudo inputs to use')
82      parser.add_argument('-smin', '--scope_min', dest='scope_min', type=int,
```

```
83                          help='Minimum  value for autogenerated data')
84      parser.add_argument('-smax', '--scope_max', dest='scope_max', type=int,
85                          help='Maximum value for autogenerated data')
86      parser.add_argument('--normalize', dest='normalize', nargs='?',type=bool,
87                          const=True,default=False, help='True for normalization')
88
89      #Performance
90      parser.add_argument('--incr-max', type=int, dest='incr_max',
91                          help='Max train size for performace iterations')
92      parser.add_argument('--incr-start', type=int, dest='incr_start',
93                          help='Initial train size for performance iterations')
94      parser.add_argument('--step', type=int, dest='step',
95                          help='Increment to add to start until max value is reached')
96      parser.add_argument('--reps', '--test-repetitions', dest='n_reps', type=int,
97          help='Number of times a time performance test is going to be executed and
                averaged')
98
99      #Hardware config
100     parser.add_argument('--gpu', dest='gpu', nargs='?',type=bool,const=True,
101                         default=False, help='Use gpu as device')
102     parser.add_argument('--nthreads', dest='nthreads', type=int,
103                         help='Number of threads to use')
104     parser.add_argument('--dtype', type=str, dest='dtype',
105                         help='Pytorch dtype options: -dtype [double|float]')
106
107     #Hyperparameter optimization
108     parser.add_argument('--lr', '--learning_rate', type=float,
109                         dest='learning_rate', help='Optimization learning rate')
110     parser.add_argument('--epochs', type=int, dest='epochs', help='Learning epochs')
111
112     #Sparse config
113     parser.add_argument('--nbatches', type=int, dest='n_batches',
114                         help='Number of train minibatches for sgp')
115     parser.add_argument('--batch-trsize', type=int, dest='batch_trsize',
116                         help='Size of the train minibatch')
117     parser.add_argument('--batch-tstsize', type=int, dest='batch_tstsize',
```

Operaciones auxiliares con tensores.

```
1  import torch as t
2  from utilities.logger import Logger
3
4  def checkMatrixProperties(matrix,name=""):
5      if(Logger.canBeLogged("Debug")):
6          if(len(name)>0):
7              name = " (" + name + ") "
8          else:
9              name = " "
10         print("")
```

```
11          Logger.logDebug("------Starting to check matrix"+\
12                  name+"properties-------",[],"checkMatrixProperties")
13          Logger.logDebug("Matrix of size: " + str(matrix.size()),[],
14                          "checkMatrixProperties")
15
16          if(isSymmetric(matrix) == False):
17              Logger.logDebug("Asymmetric matrix",[],"checkMatrixProperties")
18          else:
19              Logger.logDebug("Symmetric matrix",[],"checkMatrixProperties")
20
21          if(True in t.isnan(matrix)):
22              Logger.logDebug("Nan detected, aborting check.",[],"checkMatrixProperties
                    ")
23              return
24
25          if(isDefinitePositive(matrix)):
26              Logger.logDebug("Definite positive",[],"checkMatrixProperties")
27          else:
28              Logger.logDebug("Non Definite positive",[],"checkMatrixProperties")
29
30          try:
31              m_inv = t.inverse(matrix)
32              Logger.logDebug("Matrix has inverse",[],"checkMatrixProperties")
33
34              if(isSymmetric(m_inv)==False):
35                  Logger.logDebug("Inverse matrix is not symmetric",[],"
                        checkMatrixProperties")
36              else:
37                  Logger.logDebug("Symmetric inverse matrix",[],"checkMatrixProperties"
                        )
38
39              if(isDefinitePositive(m_inv)):
40                  Logger.logDebug("Definite positive inverse",[],"checkMatrixProperties
                        ")
41              else:
42                  Logger.logDebug("Non Definite positive inverse",[],"
                        checkMatrixProperties")
43          except:
44              Logger.logDebug("Matrix doesn't have inverse",[],"checkMatrixProperties")
45          if(matrix.size()[0] < 10 and matrix.size()[1]< 10):
46              Logger.logDebug("\n"+str(matrix))
47
48          print("")
49
50  def isDefinitePositive(matrix):
51      eigenvalues,_ = matrix.eig()
52      if(False in (eigenvalues[:,0] > 0)):
53          return False
```

```
54      return True
55
56  def isSymmetric(matrix):
57      if(matrix.size()[0] !=  matrix.size()[1]):
58          return False
59
60      for i in range(matrix.size()[1]):
61          if(False in (matrix[:][i] == matrix[i][:])):
62              return False
63      return True
64
65  def dist(x,xstar,n=1):
66      size = x.size()[1]
67      if (t.is_tensor(xstar)):
68          if(size != xstar.size()[1]):
69              Logger.logError("x and xstar have different d-features")
70          return t.cdist(x,xstar)
71      else:
72          return t.cdist(x,x)
73
74  def cross_diag(dev_size, xstar, x_size):
75      if(t.is_tensor(xstar)):
76
77          return t.zeros()
78      else:
79          return t.eye(dev_size)
80
81  def positive_definite_diag(matrix):
82      d1, d2 = matrix.size()
83      if(d1 != d2):
84          Logger.logError("Positive definite diag for non square matrix")
85
86      return matrix.tril(-1) + non_neg_(matrix.diag()).diag() + matrix.triu(1)
87
88  def non_neg_(vector):
89      return t.exp(vector)
```

Depuración de tensores.

```
1
2   def getGraph(var_grad_fn, n=0):
3       if(n == 50): exit()
4       tab="("+str(n)+")"
5       for i in range(n):
6           tab +="-"
7
8       print(tab,end="")
9       print(var_grad_fn)
10
```

```
    for b in var_grad_fn.next_functions:
        if(b[0]):
            getGraph(b[0], n+1)


def profiling(fun, params):
    with t.autograd.profiler.profile() as prof:
        fun(params)
    print(prof.key_averages().table(sort_by="self_cpu_time_total"))
```

B.1.6. Tests

```
from inference.gaussian import GaussianProcess
from inference.sparse_gaussian import SparseGaussianProcess
from covariances.rbfKernel import RBF
from covariances.sinKernel import SIN
from utilities.logger import Logger
from data.partition import RandomPartition, ExplicitPartition
from data.dataset import Dataset
from utilities.metric import MSE, RMSE, SMSE, RSMSE, Metrics
from utilities.metric import TestLogLikelihood, MLElbo
from utilities.performance import *
from utilities.input import repack
import matplotlib.pyplot as plot
import torch as t



########################################
###                                  ###
###         Dataset tests            ###
###                                  ###
########################################
#test 1
def test_autosampling_rbf(params, needPlot=True):
    tensor_params = {"dtype": params["dtype"], "device": params["device"]}
    # Create RBF kernel as generator engine for autosampling
    kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
    if("dataset_file" in params): del params["dataset_file"]
    # Generate an autosampled dataset
    dataset, c_time =  used_time(lambda: Dataset(params, generator=kernel))

    Logger.logInfo("Starting plot" )
    name = params["img_dir"] + params["test_name"]

    dataset.plot_sample(name +"_sampled", dpi= params["dpi"])

    return {"Sample generation (Gaussian) time": c_time}
```

```
37
38 #test 2
39 def test_autosampling_sin(params, needPlot=True):
40     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
41     kernel = SIN(params["l_scale"], params["noise"], params["var"], 2, tensor_params)
42     if("dataset_file" in params): del params["dataset_file"]
43     dataset, c_time =  used_time(lambda: Dataset(params, generator=kernel))
44
45     Logger.logInfo("Starting plot" )
46     name = params["img_dir"] + params["test_name"]
47
48     dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
49
50     return {"Sample generation (periodic) time": c_time}
51
52 #test 3
53 def test_autosampling_rbf_sin(params, needPlot=True):
54     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
55     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params,
           jointHyp=True)
56     kernel2 = SIN(params["l_scale"], params["noise"], params["var"], 2, tensor_params
           , jointHyp=True)
57     dataset, c_time =  used_time(lambda: Dataset(params, generator=kernel+kernel2))
58
59     Logger.logInfo("Starting plot" )
60     name = params["img_dir"] + params["test_name"]
61
62     dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
63
64     return {"Sample (+) generation time": c_time}
65
66 ########################################
67 ###                                  ###
68 ###      GP &  Partition tests       ###
69 ###                                  ###
70 ########################################
71
72 #test 4
73 def test_random_partition(params, needPlot=True):
74
75     #Generating an autosampled Dataset
76     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
77     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
78     dataset = Dataset(params, generator=kernel)
79
80     #Generating a partition with training partition of 5 tuples
81     partition, c_time = used_time(lambda: RandomPartition(dataset, size =params["
           xn_data"] , seed=2))
```

```
82      gp = GaussianProcess(partition, kernel, repack(params))
83
84      _, tr_time = gp.train()
85      _, ts_time = gp.test()
86
87      if(needPlot):
88          Logger.logInfo("Starting plot" )
89          name = params["img_dir"] + params["test_name"]
90
91          dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
92          gp.plot_posterior(name +"_posteriori", dpi= params["dpi"])
93
94      return {"Partitioning (random) time": c_time,
95              "Training time": tr_time,
96              "Testing time": ts_time}
97
98
99  #test 5
100 def test_explicit_partition(params, needPlot=True):
101     #Generating an autosampled Dataset
102     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
103     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
104     dataset = Dataset(params, generator= kernel)
105
106     #Generating a partition with training partition of 5 tuples
107     idx = t.linspace(0, params["xstarn_data"]-1, params["xn_data"]).long().tolist()
108     partition, c_time = used_time(lambda: ExplicitPartition(dataset, idx))
109
110     gp = GaussianProcess(partition, kernel, repack(params))
111
112     _, tr_time = gp.train()
113     _, ts_time = gp.test()
114
115     if(needPlot):
116         Logger.logInfo("Starting plot" )
117         name = params["img_dir"] + params["test_name"]
118
119         dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
120         gp.plot_posterior(name +"_posteriori", dpi= params["dpi"])
121
122     return {"Partitioning (explicit) time": c_time,
123             "Training time": tr_time,
124             }
125
126 ########################################
127 ###                                  ###
128 ###         GP optim tests           ###
129 ###                                  ###
```

```
130 ######################################
131 #test 6
132 def test_hyperp_optimization(params, needPlot=True):
133     #Generating an autosampled Dataset
134     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
135     #Kernel generator with random hyperparameters
136     kernel = RBF(0.5,1e-8,1,params = tensor_params)
137     dataset = Dataset(params, generator=kernel)
138 
139     #Generating a random partition
140     partition = RandomPartition(dataset, size =params["xn_data"] , seed=2)
141     #Generating a kernel for our GP
142     if(params["l_scale"] == 0.5 and params["noise"]==1e-8 and params["var"]==1):
143         Logger.logWarning("Using the same hyperparameters to generate the sample and
                to star optimizing")
144         Logger.logWarning("Use options: '-l [val] -n [val] -v [val]' to change
                defaults")
145     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
146 
147     gp = GaussianProcess(partition, kernel, repack(params))
148     h_time = gp.hyper_optim()
149     _, tr_time = gp.train()
150     _, ts_time = gp.test()
151 
152 #     Logger.logInfo("RMSE = " +\
153 #             str(MSE.calculate((gp.posterior_data["test_means"], gp.test_data["Ytest
    "]))))
154 #     Logger.logInfo("NTLL = " + str(-gp.testLogLikelihood()))
155     Logger.logInfo("Optimized parameters =>" + kernel.showParams())
156 
157     if(needPlot):
158         Logger.logInfo("Starting plot" )
159         name = params["img_dir"] + params["test_name"]
160 
161         dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
162         gp.plot_posterior(name +"_posteriori", dpi= params["dpi"])
163 
164     return {"Optimization time": h_time,
165             "Optimization/epoch time": h_time/params["epochs"]
166             }
167 
168 ######################################
169 ###                                ###
170 ###         Sparse GP tests        ###
171 ###                                ###
172 ######################################
173 #test 7
174 def test_sgp(params, needPlot=True):
```

```
175     #Generating an autosampled Dataset
176     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
177     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
178     #if("dataset_file" in params): del params["dataset_file"]
179     dataset = Dataset(params, generator=kernel)
180 
181     #Generating a partition with training partition of 5 tuples
182     partition = RandomPartition(dataset, size =params["xn_data"],
183                 selector=ExplicitPartition.batch_selector,seed=params["seed"])
184 
185     gp = SparseGaussianProcess(partition, kernel, repack(params))
186 
187     _, tr_time, m_time = gp.train()
188     _, ts_time = gp.test()
189 
190 
191     if(needPlot):
192         Logger.logInfo("Starting plot" )
193         name = params["img_dir"] + params["test_name"]
194 
195         dataset.plot_sample(name +"_sampled", dpi= params["dpi"])
196         gp.plot_posterior(name +"_predictive", dpi= params["dpi"])
197 
198     return {"Training time":       tr_time,
199             "Training/epoch time": tr_time/params["epochs"],
200             }
201 
202 
203 ################################################
204 ###                                          ###
205 ###         Quality of predictions           ###
206 ###                                          ###
207 ################################################
208 #test 8
209 def test_sgp_metrics(params, needPlot=True):
210 
211     #Import dataset, if it doesnt exist it is autosampled one with the current kernel
212     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
213     kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
214     dataset = Dataset(params, generator=kernel)
215 
216     # Partitionate data
217     partition = RandomPartition(dataset, size =params["xn_data"],
218                 selector=ExplicitPartition.batch_selector,seed=params["seed"])
219 
220     #Create an SparseGaussioanProcess
221     n = params["error_iter"]
222     error = Metrics(MSE(n), TestLogLikelihood(n), MLElbo(n))
```

```
223     gp = SparseGaussianProcess(partition, kernel, repack(params), error, params["
            normalize"])
224
225     #Joint optimization, training and test
226     _, tr_time, m_time = gp.train()
227     _, ts_time = gp.test()
228
229     #Plot error and return times
230     if(needPlot):
231         name = params["img_dir"] + params["test_name"]
232         gp.plot_error(name+"_error", dpi = params["dpi"])
233
234     return {
235             "Training time":       tr_time,
236             "Training/epoch time": tr_time/(len(gp.train_batches) * params["epochs"])
                ,
237             }
238
239
240 ################################################
241 ###                                          ###
242 ###               Performance                ###
243 ###                                          ###
244 ################################################
245
246 #test 9
247 def test_gp_performance(params):
248     def intern_f(params, needPlot=True):
249         tensor_params = {"dtype": params["dtype"],
250                          "device": params["device"]}
251         kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params
                )
252
253         #Generating an autosampled Dataset
254         dataset = Dataset(params, generator=kernel)
255
256         #Generating a partition with training partition of 5 tuples
257         partition = RandomPartition(dataset, size =params["xn_data"] , seed=2)
258         gp = GaussianProcess(partition, kernel, repack(params))
259         _, tr_time = gp.train()
260         _, ts_time = gp.test()
261
262         return {"Training time":       tr_time,
263                 "Training/epoch time": tr_time/params["epochs"],
264                 "Test/epoch time":     ts_time}
265
266     params["xn_data"] = params["incr_start"]
267     i, r = time_measurement(intern_f, params)
```

```
268
269     f = plot.figure()
270
271     tr_time =  [ i["Training time"] for i in r ]
272     tre_time = [ i["Training/epoch time"] for i in r ]
273
274
275     plot.xlabel("Train size")
276     plot.ylabel("Time (s)")
277     plot.plot(i, tr_time, 'b-', label='training time')
278     plot.plot(i, tre_time, 'g-', label='training/epoch time')
279
280     legend = plot.legend(loc="upper center", bbox_to_anchor = (0.5, 1.1), ncol=3)
281     f.savefig(params["img_dir"] + params["test_name"], dpi=params["dpi"])
282
283     return {}
284
285 #test 10
286 def test_gphyperp_performance(params):
287     params["xn_data"] = params["incr_start"]
288     i, r = time_measurement(test_hyperp_optimization, params)
289
290     f = plot.figure()
291     op_time =  [ i["Optimization time"] for i in r ]
292     plot.plot(i, op_time, 'b-', label='performance')
293
294     plot.xlabel("Train size")
295     plot.ylabel("Time (s)")
296     legend = plot.legend(loc="upper center", bbox_to_anchor = (0.5, 1.1), ncol=3)
297     f.savefig(params["img_dir"] + params["test_name"], dpi=params["dpi"])
298
299     return {}
300
301
302 def test_sgp_performance(params):
303     i, r = time_measurement(test_sgp_metrics, params,incr_tr_size)
304     f = plot.figure()
305     tr_time =  [ i["Training time"] for i in r ]
306     tre_time = [ i["Training/epoch time"] for i in r ]
307
308
309     plot.xlabel("Batch size")
310     plot.ylabel("Time (s)")
```

B.2. Script para la ejecución automática de todos los tests.

```
1 #!/bin/bash
2
```

```
3  # To check hyperparam changes, use: --log-dmask "hyperparameters" --log-dmode file
4
5  if [[ $1 == "clean" ]]; then
6          rm output/img/*
7          rm output/performance/*
8          rm output/log/*
9          return
10 fi
11
12 OUT=$(tty)
13 OPTIONS4="-l 2 -n 0.5 -v 0.1"
14 OPTIONS6="--epochs 10 --incr-max 2500"
15 OPTIONS7="--epochs 10 --incr-max 2500"
16 OPTIONS8="--epochs 10 --incr-max 2500"
17
18 #Kernel
19 python3 main.py -cfile "input/conf/default_conf.json" -t "1,2,3" $OPTIONS --n-size
       150 --gpu >> $OUT
20
21 #Autosampled Gaussian Process
22 python3 main.py -cfile "input/conf/default_conf.json" -t "4,5,6" $OPTIONS2 --gpu >>
       $OUT
23
24 #Gaussian Process
25 python3 main.py -cfile "input/conf/physicochemical_pconf.json" --x-size 2500 -t "4,5"
        $OPTIONS3 --gpu >> $OUT
26 python3 main.py -cfile "input/conf/physicochemical_pconf.json" --x-size 2500 -t "6"
       $OPTIONS4 --epochs 10 --gpu >> $OUT
27 python3 main.py -cfile "input/conf/physicochemical_pconf.json" --x-size 2500 -t "6"
       $OPTIONS4 --gpu --epochs 100 >> $OUT
28 python3 main.py -cfile "input/conf/physicochemical_pconf.json" --x-size 2500 -t "6"
       $OPTIONS4 --gpu --epochs 1000 >> $OUT
29
30 #Hyperparameter variations in GP posterior
31 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 2
32 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 4
33 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 1
       -n 0.5
34 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 1
       -n 1
35 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 1
       -v 0.5
36 python3 main.py -cfile "input/conf/default_conf.json" -t "4" --n-size 150 --gpu -l 1
       -v 2
37
38 #Sparse Gaussian Process
39 #python3 main.py -cfile "input/conf/default_conf.json" -t "7" $OPTIONS5 --gpu >> $OUT
40 python3 main.py -cfile "input/conf/year_prediction_conf.json" -t "7,8" $OPTIONS5 --
```

```
    gpu >> $OUT
41  python3 main.py -cfile "input/conf/default_conf.json" -t "7" --gpu --error-iter 25 --
    x-size 100 --n-size 200 --u-size 25 --batch-trsize 40 --batch-tstsize 60 --epochs
     1000
42
43  #Performance tests for section 5.2.5
44  python3 main.py -cfile "input/conf/physicochemical_pconf.json" -t "10,11" -l 2 -n 0.5
    -v 0.1 --epochs 10 --incr-max 4000
45  python3 main.py -cfile "input/conf/physicochemical_pconf.json" -t "10,11" -l 2 -n 0.5
    -v 0.1 --epochs 10 --incr-max 4000 --gpu
46  python3 main.py -cfile "input/conf/year_prediction_pconf.json" -t "11" -l 2 -n 0.5 -v
    0.1 --epochs 10 --incr-max 5000
47  python3 main.py -cfile "input/conf/year_prediction_pconf.json" -t "11" -l 2 -n 0.5 -v
    0.1 --epochs 10 --incr-max 5000 --gpu
48
49  #GP vs SGP Performance for physicochemical dataset
50  python3 main.py -cfile "input/conf/physicochemical_pconf.json" -t "10,11" $OPTIONS4
    $OPTIONS6 >> $OUT
51  python3 main.py -cfile "input/conf/physicochemical_pconf.json" -t "10,11" $OPTIONS4
    $OPTIONS6 --gpu >> $OUT
52
53  #Sparse Gaussian Process Performance for large datasets
54  python3 main.py -cfile "input/conf/year_prediction_pconf.json" -t "11" $OPTIONS7 >>
    $OUT
55  python3 main.py -cfile "input/conf/year_prediction_pconf.json" -t "11" $OPTIONS7 --
    gpu >> $OUT
56
57
58  #python3 main.py -cfile "input/conf/default_conf.json" -t "x" --x-size 3 --n-size 150
    --gpu -l 2 -v 0.5
```

B.3. Ficheros de configuración para la ejecución de los tests.

Fichero de configuración por defecto.

```
1  {
2          "tests":"1,2,3,4,5,6,7",
3          "gpu": "false",
4          "colorless": "false",
5          "seed": "5",
6          "nthreads": 1,
7
8          "xn_data": 1000,
9          "xstarn_data": 1500,
10         "u_data":100,
```

```
        "scope_min": -30,
        "scope_max": 30,

        "dtype": "double",

        "l_scale": 0.5,
        "var": 1,
        "noise":0.00000001,
        "learning_rate": 0.01,
        "epochs":100,

        "dataset_file": "Empty",
        "performance_dir": "output/performance/",
        "img_dir": "output/img/",
        "log_dir": "output/log/",
        "n_reps": 1,

        "dpi":200,
        "step":250,
        "incr_max": 1250,
        "incr_start": 250,

        "log_level": 5,
        "log_mode": "desc",
        "log_dmode": "text",
        "log_dmask": "Hyperparameter.setValue, ",

        "n_batches": 0,
        "batch_trsize": 500,
        "batch_tstsize":500,
        "error_iter": 500
}
```

Ficheros de configuración para el dataset Airlines.

```
{
        "tests":"10",
        "gpu": "false",
        "colorless": "false",
        "seed": "5",
        "nthreads": 1,
        "normalize": true,

        "xn_data": 2117068,
        "xstarn_data": 10000,
        "u_data":100,
        "scope_min": -5,
        "scope_max": 5,

```

```
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
19          "noise":0.00000001,
20          "learning_rate": 0.01,
21          "epochs":100,
22
23          "dataset_file": "input/datasets/airlines.txt",
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 10000,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "Hyperparameter.setValue, buffered, MSE, RMSE,
                test_loglikelihood,elbo ",
38
39          "n_batches": 1000,
40          "batch_trsize": 0,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

```
1   {
2           "tests":"10",
3           "gpu": "false",
4           "colorless": "false",
5           "seed": "5",
6           "nthreads": 1,
7           "normalize": true,
8
9           "xn_data": 2117068,
10          "xstarn_data": 10000,
11          "u_data":100,
12          "scope_min": -5,
13          "scope_max": 5,
14
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
```

```
19          "noise":0.00000001,
20          "learning_rate": 0.01,
21          "epochs":10,
22
23          "dataset_file": "input/datasets/airlines.txt",
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 7000,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "",
38
39          "n_batches": 0,
40          "batch_trsize": 500,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

Ficheros de configuración para el dataset YearPredictionMSD.

Configuración para el dataset extraído del repositorio UCI

```
1   {
2           "tests":"10",
3           "gpu": "false",
4           "colorless": "false",
5           "seed": "5",
6           "nthreads": 1,
7           "normalize": true,
8
9           "xn_data": 505345,
10          "xstarn_data": 10000,
11          "u_data":100,
12          "scope_min": -5,
13          "scope_max": 5,
14
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
19          "noise":0.00000001,
```

```
20          "learning_rate": 0.01,
21          "epochs":100,
22
23          "dataset_file": "input/datasets/YearPredictionMSD.txt",
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 10000,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "Hyperparameter.setValue, buffered, MSE, RMSE,
                test_loglikelihood,elbo ",
38
39          "n_batches": 1000,
40          "batch_trsize": 0,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

```
1   {
2           "tests":"10",
3           "gpu": "false",
4           "colorless": "false",
5           "seed": "5",
6           "nthreads": 1,
7           "normalize": true,
8
9           "xn_data": 505345,
10          "xstarn_data": 10000,
11          "u_data":100,
12          "scope_min": -5,
13          "scope_max": 5,
14
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
19          "noise":0.00000001,
20          "learning_rate": 0.01,
21          "epochs":10,
22
23          "dataset_file": "input/datasets/YearPredictionMSD.txt",
```

```
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 7000,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "",
38
39          "n_batches": 0,
40          "batch_trsize": 500,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

Ficheros de configuración para el dataset Physicochemical.

Configuración para el dataset extraído del repositorio UCI

```
1   {
2           "tests":"9,10",
3           "gpu": "false",
4           "colorless": "false",
5           "seed": "5",
6           "nthreads": 1,
7           "normalize": true,
8
9           "xn_data": 40730,
10          "xstarn_data": 45730,
11          "u_data":100,
12          "scope_min": -5,
13          "scope_max": 5,
14
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
19          "noise":0.00000001,
20          "learning_rate": 0.01,
21          "epochs":10,
22
23          "dataset_file": "input/datasets/physicochemical.txt",
```

```
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 6000,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "",
38
39          "n_batches": 0,
40          "batch_trsize": 500,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

Ficheros de configuración para el dataset PowerPlant.

Configuración para el dataset extraído del repositorio UCI

```
1   {
2           "tests":"9,10",
3           "gpu": "false",
4           "colorless": "false",
5           "seed": "5",
6           "nthreads": 1,
7           "normalize": true,
8
9           "xn_data": 8568,
10          "xstarn_data": 1000,
11          "u_data":100,
12          "scope_min": -5,
13          "scope_max": 5,
14
15          "dtype": "double",
16
17          "l_scale": 0.5,
18          "var": 1,
19          "noise":0.00000001,
20          "learning_rate": 0.01,
21          "epochs":10,
22
23          "dataset_file": "input/datasets/power_plant.txt",
```

```
24          "performance_dir": "output/performance/",
25          "img_dir": "output/img/",
26          "log_dir": "output/log/",
27          "n_reps": 1,
28
29          "dpi":200,
30          "step":250,
31          "incr_max": 8500,
32          "incr_start": 500,
33
34          "log_level": 4,
35          "log_mode": "desc",
36          "log_dmode": "text",
37          "log_dmask": "",
38
39          "n_batches": 0,
40          "batch_trsize": 500,
41          "batch_tstsize":500,
42          "error_iter": 500
43  }
```

B.4. Código R que implementa la Distribución Gaussiana.

```
1  #Definicion de la funcion de densidad
2  DistribucionGaussiana <- function(x, med, sig) {
3      1 / sqrt(2 * 3.14 * sig^2) * exp(- 1 / (2 * sig^2) * (x - med)^2)
4  }
5
6
7  n_data <- 100
8
9  x <- seq(-5, 5, length = n_data)
10 #media <- mean(x)
11 #varianza <- var(x)
12
13 #Generar el grafico
14 plot(x, DistribucionGaussiana(x, 0, 1), type = "l", col = "blue")
```

B.5. Código Python para dibujar muestras del prior y posterior de un GP.

Este código hay que añadirlo al diccionario de funciones de main.py para poder usar las opciones de input.

```
from inference.gaussian import GaussianProcess
from inference.sparse_gaussian import SparseGaussianProcess
from covariances.rbfKernel import RBF
from covariances.sinKernel import SIN
from utilities.logger import Logger
from data.partition import RandomPartition, ExplicitPartition
from data.dataset import Dataset
from utilities.metric import MSE, RMSE, SMSE, RSMSE, Metrics
from utilities.metric import TestLogLikelihood, MLElbo
from utilities.performance import *
from utilities.input import repack
import matplotlib.pyplot as plot
import torch as t



def test_generatesamples(params):

    NUMERO_SAMPLES = 3 # supported plot (colors) for 5 samples max
    path = params["img_dir"] + params["test_name"]
    dpi=params["dpi"]
    flag = True

    tensor_params = {"dtype": params["dtype"], "device": params["device"]}
    # Create RBF kernel as generator engine for autosampling
    kernel = RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
    if("dataset_file" in params): del params["dataset_file"]
    # Generate autosampled datasets
    datasets = [Dataset(params, generator=kernel) for i in range(NUMERO_SAMPLES)]

    #Choose the observations for the inference problem
    idx = t.randperm(params["xstarn_data"]).tolist()[:params["xn_data"]]
    partition = ExplicitPartition(datasets[0],idx)

    #Create GPs
    gp = GaussianProcess(partition, kernel, repack(params))

    #Plot autosampled data
    f = plot.figure()
    plot.title("Sample")
    colors = ["c-","y-", "m-","g-", "b-"]
    for i in range(NUMERO_SAMPLES):
        x, y = datasets[i].getData()
        plot.plot(x.cpu(), y.cpu().detach().numpy(), colors[i], label='Sample '+str(i
            ))
    legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,-0.05), ncol=3)
    f.savefig(path+"_prior", dpi=dpi, bbox_extra_artists = [legend],bbox_inches="
        tight")
```

```
47
48      #Train, test and plot posterior
49      _, tr_time = gp.train()
50      _, ts_time = gp.test(full_cov=True)
51
52      xtrain, ytrain, xtest, ytest, _, _ =  gp.get_repr_data()
53      means, full_cov = gp.get_posterior_data()
54      deviations = 2*t.sqrt(t.diag(full_cov))
55      gp_v = t.cholesky(full_cov)
56      gp_m = means
57
58      f = plot.figure()
59      plot.title("Sample posteriors")
60      rvector = [ t.normal(0,1, (1,gp_m.size()[0]), dtype=params["dtype"])[0] for i in
            range(NUMERO_SAMPLES) ]
61      ysamples = [ gp_m  + t.matmul(rvector[i].unsqueeze(0), gp_v.t()).squeeze(0)  for
            i in range(NUMERO_SAMPLES) ]
62      for i in range(NUMERO_SAMPLES):
63          plot.plot(xtest, ysamples[i], colors[i], label='sample'+str(i))
64      if(flag == True):
65          #observations
66          plot.plot(xtrain, ytrain,'r.', label='observations')
67          #standard deviations
68          p_dev_neg = means - deviations
69          p_dev_plu = means + deviations
70          plot.fill_between(xtest, p_dev_neg, p_dev_plu, color="g",edgecolor=None,alpha
                =0.2, label='covariance')
71          flag = False
72      legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,-0.05), ncol=3)
73      f.savefig(path+"_posterior", dpi=dpi, bbox_extra_artists = [legend],bbox_inches="
            tight")
74
75      return {}
```

B.6. Código Python para dibujar múltiples muestras con distintos hiperparámetros.

Este código hay que añadirlo al diccionario de funciones de main.py para poder usar las opciones de input e invocar el programa de la siguiente forma: python3 main.py -cfile "input/conf/default_conf.json"-t "x"), donde x es el número del test que se haya escogido.

```
1  from inference.gaussian import GaussianProcess
2  from inference.sparse_gaussian import SparseGaussianProcess
3  from covariances.rbfKernel import RBF
4  from covariances.sinKernel import SIN
```

```
5  from utilities.logger import Logger
6  from data.partition import RandomPartition, ExplicitPartition
7  from data.dataset import Dataset
8  from utilities.metric import MSE, RMSE, SMSE, RSMSE, Metrics
9  from utilities.metric import TestLogLikelihood, MLElbo
10 from utilities.performance import *
11 from utilities.input import repack
12 import matplotlib.pyplot as plot
13 import torch as t
14
15 #Important load this code as a main.py test to use input functionality
16
17 def test_generatePriors(params):
18     NUMERO_SAMPLES = 3 # supported plot (colors) for 5 samples max
19     path = params["img_dir"] + params["test_name"]
20     dpi=params["dpi"]
21
22     tensor_params = {"dtype": params["dtype"], "device": params["device"]}
23     # Create RBF kernel as generator engine for autosampling
24     kernels = [RBF(params["l_scale"], params["noise"], params["var"], tensor_params)
           for i in range(NUMERO_SAMPLES)]
25     if("dataset_file" in params): del params["dataset_file"]
26     # Generate autosampled datasets
27     datasets = [Dataset(params, generator=kernels[i]) for i in range(NUMERO_SAMPLES)]
28
29     #Plot autosampled data
30     f = plot.figure()
31     plot.title("Sample")
32     colors = ["c-","y-", "m-","g-", "b-"]
33     for i in range(NUMERO_SAMPLES):
34         x, y = datasets[i].getData()
35         plot.plot(x.cpu(), y.cpu().detach().numpy(), colors[i], label='Sample '+str(i
               ))
36     legend = plot.legend(loc="upper center",bbox_to_anchor = (0.5,-0.05), ncol=3)
37     plot.ylim([-3.5,3.5])
38     f.savefig(path+"_prior", dpi=dpi, bbox_extra_artists = [legend],bbox_inches="
           tight")
39
40     return {}
```

C

Estructura de directorios

La estructura del paquete de software desarrollado se puede ver en la figura C.1.

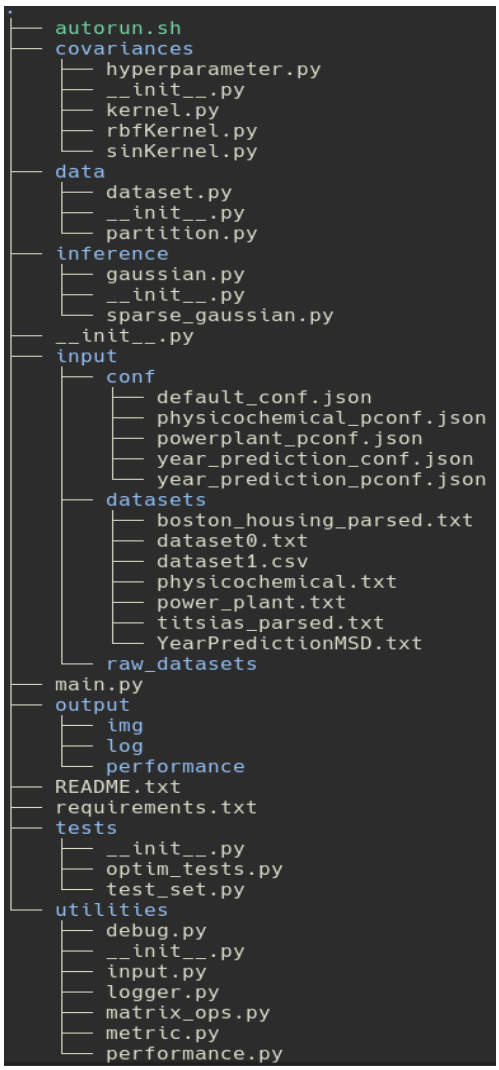


Figura C.1: Árbol de directorios del proyecto generado con la herramienta tree en bash, GNU/Linux.

D

CONFIGURACIÓN DE LA ENTRADA

En este apéndice vamos a detallar la especificación de la configuración de la entrada. En concreto, en las tabla D.2, D.4,D.6 y D.8 se exponen las opciones de configuración por línea de comandos y por fichero en formato json.

Json	**Bash**	**Descripción**	**Valor**
tests	-t	Ejecutar los tests especificados	String con el numero de los tests separados por comas a ejecutar
gpu: <valor>	--gpu	Habilitar el uso de GPU para computación	["true"|"false"]
colorless: <valor>	--colorless	Todo el output del color por defecto de la consola	["true"|"false"]
seed: <valor>	--seed <valor>	Valor de la inicialización de la semilla	Entero
nthreads: <valor>	--nthreads <valor>	Numero de threads para la ejecución	Entero
xn_data <valor>	--x-size <valor>	Número de instancias (tamaño) del conjunto de entrenamiento	Entero

Tabla D.2: En esta tabla se detalla la primera parte de las diferentes opciones de entrada.

Json	Bash	Descripción	Valor
xstarn_data	--n-size	Numero de instancias (tamaño) de todo el dataset auto-generado	Entero
un_data	--u-size	Numero de instancias (tamaño) del conjunto aproximado	Entero
scope_min: <valor>	-smin <valor>	Para conjuntos auto-generados es la cota mínima	Decimal
scope_max: <valor>	-smax <valor>	Para conjuntos auto generados es la cota máxima	Decimal
dtype: <valor>	–dtype <valor>	Tipo de los tensores pytorch a utilizar	["double"| "float"]
l_scale: <valor>	-l <valor>	Valor length scale de kernel	Decimal positivo
var: <valor>	-v <valor>	Valor de la varianza (amplificador kernel)	Decimal positivo
noise: <valor>	-n <valor>	Valor del ruido añadido al kernel	Decimal positivo
learning_rate: <valor>	--learning-rate <valor>	Tamaño del step a la hora de optimizar los hiperparámetros	Decimal positivo
epochs: <valor>	--epochs <valor>	Número de épocas a optimizar	Decimal
-	--cfile <valor>	Ruta al fichero con la configuración del programa en json	string con ruta al fichero

Tabla D.4: En esta tabla se detalla la segunda parte de las diferentes opciones de entrada.

Json	Bash	Descripción	Valor
dataset_file: <valor>	--dfile <valor>	Ruta al fichero con el dataset en formato csv	string con ruta al fichero
performance_dir: <valor>	--pdir <valor>	Ruta a la carpeta para almacenar registros de rendimiento	string con ruta al fichero
img_dir: <valor>	--idir <valor>	Ruta a la carpeta para almacenar imágenes	string con ruta a la carpeta
log_dir: <valor>	--ldir <valor>	Ruta a la carpeta para almacenar logs de rendimiento	string con ruta a la carpeta
dpi: <valor>	--dpi	dpi de las imágenes generadas	Entero
step: <valor>	--incr	Tamaño de la variación de la entrada a la hora de medir el rendimiento	Entero
incr_max: <valor>	--incr-max <valor>	Tamaño máximo de la entrada a la hora de medir el rendimiento	Entero
incr_start: <valor>	--incr-start <valor>	Tamaño inicial de la entrada a la hora de medir el rendimiento	Entero
log_level: <valor>	--log-level <valor>	Nivel de log	[0-5]
log_mode: <valor>	--log-mode <valor>	Modo de interpretar el nivel de log	["fixed"|"desc"]
log_dmode <valor>	--log-dmode <valor>	Elegir salida de los logs de depuración	["text"|"file"]

Tabla D.6: En esta tabla se detalla la tercera parte de las opciones de entrada.

Json	Bash	Descripción	Valor
log_dmask: <valor>	--log-dmask <valor>	Máscara para filtrar logs de depuración	string de elementos de la tabla separados por comas
n_batches: <valor>	--nbatches <valor>	Número de batches de entrenamiento por época	Entero
batch_trsize: <valor>	--batch-trsize <valor>	Tamaño del batch de entrenamiento	Entero
batch_tstsize: <valor>	--batch-tstsize <valor>	Tamaño del batch de test	Entero
error_iter: <valor>	--error-iter <valor>	Numero de batches que se tienen que computar para calcular las métricas	Entero

Tabla D.8: En esta tabla se detalla la tercera parte de las opciones de entrada.

E

Detalles del sistema de logging

En la tabla E.1, podemos ver un subconjunto de las opciones de entrada que nos van a hacer más fácil interactuar con el sistema de logging.

Json	Bash	Valor	
1	log_level <valor>	−−log−level <valor>	[0-5]
2	log_mode <valor>	−−log−mode <valor>	["desc"| "fixed"]
3	log_dmask <valor>	−−log−dmask <valor>	[mascara]
4	log_dmode <valor>	−−log−dmode <valor>	["file"| "text"]
5	colorless <valor>	−−colorless	["true"| "false"]

Tabla E.1: En esta tabla se algunas de las diferentes opciones de logging existentes.

El primer comando sirve para elegir el nivel de log de acuerdo a la lista anterior, el valor 0 se utilizará para suprimir el output. El segundo comando es para elegir si solo se quiere la salida de ese nivel de log ("fixed") o si se quiere todo la salida de ese nivel y los inferiores ("desc"). Al contrario que el resto de logs, los logs de depuración pueden aparecer dentro de bucles para ver como evoluciona un determinado valor, esto hace necesario el tercer comando, que se utiliza para filtrar solo aquellos valores que necesitemos, la lista de posibles valores se encuentra en el cuadro E.1. El cuarto comando servirá para guardar la salida de depuración en el fichero especificado en los ficheros de configuración. Finalmente, el sistema de logs tiene habilitado por defecto la capacidad para colorear la salida estándar de acuerdo al nivel de log. Esto facilita el seguimiento visual; sin embargo, si se quiere redirigir la salida a un fichero o ejecutar el código en sistemas con terminales de funcionalidad limitada, puede ser conveniente suprimir el color, esto se puede hacer mediante el último comando.

Hyperparam.getValue, Kernel.process, Kernel.marginal_like, Kernel.hyperp_optimization, GaussianProcess.means, GaussianProcess.variances, SparseGaussianProcess.Eq, iter_time, SparseGaussianProcess.kl, elbo, MSE*, RMSE*, RSMSE*, SMSE*, test_loglikelihood, Logger.logDebug, checkMatrixProperties, checkMatrixProperties, negativeML, hyperparameters, Kernel.kernel, buffered, Metrics.plot, Metric.compute.*

Cuadro E.1: En este cuadro se presentan las diferentes opciones de mascara para la depuración. Se debe generar un string con todas las opciones deseadas, separadas por comas, para la entrada del programa. Como datos adicionales, las opciones marcadas con * deberán ir acompañadas de la opción 'buffered', y la de *hyperparameters* solo se podrá habilitar para un GP standard.


UAM
UNIVERSIDAD AUTONOMA
DE MADRID